Universidad de Guanajuato
División de Ingenierías
DEPARTAMENTO DE INGENIERÍA CIVIL

Diseño, Análisis y Optimización Estructural de Edificaciones Paramétricas

---

Tesis de licenciatura que se presenta para obtener el grado de:
Ingeniero Civil

Eugenio José Muttio Zavala

Director de Tesis:
Dr. Salvador Botello Rionda

Agosto 2017

Revisores

Director de Tesis: Dr. Salvador Botello Rionda

1. Revisor: Mc. Humberto Esqueda Oliva

2. Revisor: Dr. Jesús Gerardo Valdés Vázquez

3. Revisor: Dr. Iván Cruz Aceves

Día de la defensa: 18 de Agosto de 2017

Firma del Director del jurado : _____

A mis Padres

Agradecimientos

El desarrollo de este trabajo de titulación representa la culminación de una etapa muy importante de mi vida, ya que el trasfondo de lo presentado en esta tesis solo pudo resultar a partir de la motivación que me brindaron muchas personas que a continuación mencionaré debido a su gran participación en mi formación académica.

Durante el segundo semestre de la licenciatura, tomé el curso de *"Lenguajes de Programación"*, en la que gracias a la gran forma de enseñar del maestro Jose Luis Alonzo, encontré especial interés en la automatización de procesos mediante una computadora. Sin embargo, la clase en la que realmente pude conocer sobre la aplicabilidad de la programación en la Ingeniería fue *"Métodos Numéricos"*. Todavía recuerdo el proyecto final donde, en conjunto con mi equipo, tuvimos que resolver una ecuación diferencial que describía la aceleración de un sismo, con el objetivo de analizar el comportamiento de un contenedor de agua. Por lo que decidimos realizar un programa en lenguaje C utilizando el método "Runge Kutta". Desde que tomé esa clase, me di cuenta que el área en la que desearía especializarme en un futuro estaría altamente influenciada por el análisis numérico aplicado a la ingeniería.

Por suerte el M.C. Humberto Esqueda, quien fue el maestro de Métodos Numéricos, me invito a formar parte de un grupo de investigación dentro de la facultad, denominado "Aula CIMNE". Dentro de este grupo aprendí mucho más sobre programación y técnicas avanzadas utilizando herramientas computacionales. Por lo que le agradezco de manera muy especial a Humberto, ya que gracias a él es que pude incursionar en esta área tan grandiosa que son los Métodos Numéricos aplicados en Ingeniería. Dentro de este grupo conocí al ahora M.C. Jacob Salazar, quien gracias a su siempre positiva motivación me ayudó a formar un pensamiento emprendedor y enérgico. Y supongo que al ver mi compromiso con el Aula CIMNE, decidió invitarme a realizar un proyecto en conjunto, por lo que le agradezco su valentía al querer involucrarme en una etapa tan temprana de mi formación. Pero aun más que eso, le agradezco la amistad que hemos conservado desde entonces.

Fue durante la planeación del proyecto en conjunto con Jacob, cuando él mismo me presentó con el Dr. Salvador Botello, quien es el director de este trabajo de tesis. Y que además de esto, ha sido un asesor excelente durante todo este tiempo en cuanto a la formación que deseo obtener. Sin

su asesoría, jamás hubiera pensado en participar en tantos cursos complementarios, escuelas de métodos numéricos y computación e incluso ser parte de congresos de distinta índole. Por ello, le agradezco infinitamente todo el esfuerzo que ha dado para incrementar mi formación y por consiguiente que este trabajo se haya podido realizar. Muchas gracias por sus consejos y la ayuda que me ha brindado para lograr mis metas. También le doy gracias a los sinodales por brindar su tiempo para revisar este trabajo, de manera especial agradezco al Dr. Iván Cruz de CIMAT, quien se animó a formar parte de este tribunal siendo externo a la facultad de Ingeniería Civil. Asimismo le agradezco mucho al M.I. Maximino Tapia, quien sembró la idea inicial de este proyecto y pudo ver su crecimiento durante este tiempo, sin él este trabajo no hubiera surgido de tal manera.

Muy especialmente quiero agradecer a algunos profesores de la División de Ingenierías, quienes han sido pieza fundamental en mi desarrollo profesional. En primer lugar al Dr. Julio César Leal Vaca quien fue mi tutor académico, al profesor Francisco Mirabal quien me transfirió su pasión por las matemáticas, al Ing. Carlos Arnold quien me ayudó a formar el coraje para resolver problemas, al Dr. Jesús Gerardo Valdés quien por sus clases desarrolló en mi una pasión por la investigación y es admirable por ser un ejemplo como persona, al Dr. Luis Enrique Mendoza que si no fuera por su apoyo, todo lo realizado no hubiera sido posible. También quiero agradecer a la Dra. Mabel Mendoza, al M.C. Francisco Luna, al Ing. Luis Eduardo Calderón y a muchos más que dieron un ejemplo a seguir.

Quiero agradecer a mi buen amigo Ramón "El Cruck" López, quien fue testigo y pudo experimentar junto conmigo todos los obstáculos necesarios para finalizar un programa de licenciatura, siempre de manera muy agradable y con apoyo mutuo. Además de estar al pie del cañon durante los proyectos y objetivos que nos propusimos, así como las diversas experiencias que enfrentamos. También agradezco a Aldo "El Guapetón" Lugo con quien pude hacer una buena mancuerna de estudio y trabajo durante la carrera de Ingeniería Civil, y por su apoyo durante los momentos desafiantes al final de ésta. También debo de agradecer a los compañeros de generación con quienes comencé este trayecto y quiénes sirvieron como apoyo grupal para sobrellevar los retos más difíciles, en especial agradezco a Olaf Ruiz, Alfredo Romero, Emmanuel Morales, Daniel Quintero, Octavio Dueñez, Agustín Nuñez y a todos en general. También agradezco al equipo del Aula CIMNE, con quienes compartimos retos diferentes a las actividades de la carrera pero que ayudaron a formarnos con nuevas habilidades, de manera especial a Eduardo Morales, Axel Tinajero, César Ruiz, Ernesto Ortega y a los nuevos integrantes que espero encuentren motivación para continuar con este camino.

Pero más allá de la formación académica, quiero agradecer a las personas más importantes en mi vida, quienes son la razón por las que pude terminar la licenciatura en Ingeniería Civil. En primer lugar agradezco a mis padres, quienes principalmente me dieron un apoyo incondicional

en las decisiones que he tomado y en segundo lugar porque me dieron el apoyo económico suficiente. Pero más importante aún, me enseñaron que en el trabajo es fundamental la responsabilidad y el compromiso, es por todo lo anterior que estaré siempre agradecido con ustedes. A mi madre, que siempre estuvo cerca de mi brindándome alivio y sustento, pero también por ayudarme con su "despensa o itacate semanal". A mi padre, por enseñarme a tomar decisiones y sobrellevar las consecuencias con cada una de éstas, y también a cargar siempre con el "sentido común" sin importar en qué situación me encuentre. A mi hermana, por darme su apoyo y un buen chiste o "meme" para comenzar el día. Pero también por darme consejos y guiarme cuando más lo necesité. También agradezco a mis abuelos, tíos y primos con quienes sé que puedo contar.

Finalmente quiero agradecer a la persona que ha estado presente en cada decisión que he tomado, y que en conjunto hemos aprendido lo maravilloso que es este camino llamado vida. Juntos hemos trascendido en distintas etapas y si no fuera por tu paciencia, comprensión y apoyo, esto no hubiera sido posible. Muchas gracias Sinaí Ochoa, eres mi motor, tú me das la fuerza necesaria para seguir adelante.

¡Gracias!

Resumen

La tendencia actual en la arquitectura es la búsqueda de espacios eficientes mediante un diseño innovador y con mínimos recursos energéticos. Debido a esto, la complejidad de los proyectos y los trabajos realizados por los ingenieros civiles crece exponencialmente. Se ha observado que la logística de proyectos se vuelve cada vez más difícil, por ello el tiempo empleado para cada etapa de planeación debe ser óptima. *La complejidad del diseño y análisis estructural se está volviendo un problema de conceptualización geométrica, de múltiple solución y de comportamientos estructurales sensibles a las propuestas cambiantes.* Además de esto, la propuesta de soluciones estructurales requieren de un proceso de cálculo numérico que puede ser costoso computacionalmente, dependiendo de la complejidad de la estructura. En la práctica profesional, las oficinas de Ingeniería afrontan modificaciones constantes en los proyectos, desde cuestiones arquitectónicas, cambios en los materiales o alteraciones en el procedimiento constructivo, por lo que hay que realizar iteraciones en puntos clave del proceso. La parametrización de procesos, ayuda a reducir los ciclos debido a los cambios que ocurren durante la ejecución de los trabajos, lo que repercutirá en un ahorro de tiempo y un óptimo control del proyecto.

Los ingenieros estructurales deben buscar nuevas alternativas a los procesos tradicionales de estructuración. Aún con las herramientas computacionales existentes en el mercado, aparecen puntos específicos en el proceso de diseño donde se puede optimizar en algún sentido, y esto tiene un impacto en el costo de **esfuerzo − tiempo − dinero** considerable, es justamente en estas etapas donde se puede obtener una estrategia que disminuya el tiempo de realización.

El diseño paramétrico consiste en la utilización de algoritmos para generar relaciones geométricas que permitan no sólo llegar a una propuesta, sino dar origen a todo el rango de posibles soluciones, controlados por un conjunto de parámetros que pueden determinarse por el proyectista. El presente trabajo tiene por objetivo mostrar el desarrollo de una aplicación computacional que permita definir

parámetros para estructurar una edificación, que pueda obtener un campo de soluciones en poco tiempo, obtener un propuesta óptima en cuanto a su resistencia mecánica, su disposición topológica y que satisfaga las necesidades arquitectónicas solicitadas.

Organización de los capítulos

El presente trabajo cuenta con cinco capítulos esencialmente y tres apéndices, los cuales están organizados de la siguiente manera:

Introducción

Se da un panorama general del procedimiento tradicional utilizado ampliamente en las oficinas de Ingeniería Estructural. Además se describe la motivación de realizar este trabajo y los retos que existen, se presenta el objetivo general y se detalla en objetivos específicos.

Parametrización Geométrica

Este capítulo comienza con una reseña histórica del trazado con fines de planeación en la industria de la construcción, comenzando con el dibujo en un lienzo, como lo es el papel, para posteriormente comenzar con el desarrollo del sistema de comunicación gráfica computacional y por último describir las herramientas actuales utilizadas por la Arquitectura y la Ingeniería. Posterior a esto, se explica a detalle los conceptos de ***Parametrización***, y la programación de sistemas paramétricos utilizando el paradigma conocido como *"Data Flow Computing"*. Se continua con la medición de flexibilidad en un modelo paramétrico mediante programación visual y el se termina el capítulo detallando las funcionalidades del software utilizado.

Programación de Módulos Paramétricos

En este capítulo se explica a manera general el funcionamiento de cada algoritmo dentro del conjunto de módulos que se programaron para la generación de la estructura, la distribución de cargas, el análisis estructural y la verificación de resistencias. La organización de secciones va de acuerdo a como se debe conectar los nodos dentro de la programación visual por lo que es altamente recomendable

utilizar el apéndice B como referencia al estar leyendo este capítulo, de esta manera se tendrá una mejor comprensión del código. Además, este capítulo tiene la peculiaridad de que se explican los pasos de la programación pero además se tienen secciones teóricas, recomendaciones y obstáculos que se tuvieron mientras fueron apareciendo, con motivo de explicar el funcionamiento del código, es por ello que esté capítulo quizá sea la parte central de este trabajo.

Resultados

Se presentan las mediciones de flexibilidad para tener una referencia, aunque no exacta, del desempeño de los módulos programados. También se generan diversos modelos como referencia de la facilidad de uso para realizar todo un proceso de análisis y diseño estructural en edificaciones.

Conclusiones

Se termina este trabajo hablando de los beneficios obtenidos gracias al programa realizado, además se da referencia al trabajo futuro, "abriendo camino" a continuar con este proyecto para motivar a más personas a implementar estas herramientas en procesos que requieren de una reducción de tiempo en su ejecución.

Apéndices

Este trabajo cuenta con tres apéndices, el primero tiene contenido el código en *Python* de cada módulo programado. El segundo es un manual de usuario para la implementación de la red en programación visual para hacer uso de esta herramienta. El último capítulo es una extensión del apéndice A, debido a que se detalla la implementación de funciones en programación visual para tener módulos que permitan mostrar los resultados de cada etapa de funcionamiento.

Índice general

Índice de figuras xi

Índice de tablas xviii

1. Introducción **5**
1.1. Motivación 5
1.1.1. Procedimiento Tradicional en la Ingeniería Estructural . 6
1.1.2. Principales Problemas en el Proceso 9
1.1.3. Solución Propuesta 12
1.2. Objetivo 16
1.2.1. Objetivos Específicos 16

2. Parametrización Geométrica **19**
2.1. Primeros Avances en la Modelación Geométrica Computacional 19
2.1.1. Sistemas de Modelación de Sólidos 24
2.1.2. Modelación Mediante Parámetros 26
2.2. Diseño Paramétrico Computacional 29
2.2.1. Programación en el Diseño Paramétrico 34
2.2.2. Eficiencia de los Modelos Paramétricos 48
2.3. **Dynamo:** Programación Visual para el Diseño 52
2.3.1. ¿ Qué es **Dynamo**? 53
2.3.2. Anatomía de un Lenguaje de Programación Visual . . . 56
2.3.3. Diseño Computacional Geométrico 59

3. Programación de Módulos Paramétricos **65**
3.1. Modelos de Edificaciones Paramétricas 65
3.1.1. Edificación Rectangular Mediante DAG 66
3.1.2. *Torre Mayor*: Modelado Mediante Parámetros 75
3.2. Intersección de Sólido Tridimensional Mediante Planos 85
3.2.1. Programación en Paralelo 87
3.2.2. Curvas Perimetrales 94
3.3. Configuración Estructural Basada en una Retícula Base 96
3.3.1. Obtención de Superficies para Losas 101
3.3.2. Cálculo de Centroides para Losas Poligonales 103

3.3.3. Liberación de Memoria en Python 105
3.3.4. **¿Porqué liberar memoria si estamos usando Python?** 107
3.4. Propiedades de los Materiales y Secciones Transversales 109
3.5. Generador de Geometría Estructural 111
3.5.1. Structure Generator 112
3.5.2. Estructuración Externa Mediante Intersección de Niveles 115
3.5.3. Estructuración Interna 121
3.5.4. Estructuración Reticular Externa e Interna 125
3.6. Fuerzas Involucradas en Edificaciones 133
3.6.1. Metodología de Distribución de Fuerzas en Losas 133
3.6.2. Implementación en Código de la Distribución de Fuerzas en Losas 141
3.7. Análisis Estructural 149
3.7.1. Breve Historia del Análisis de Estructuras 149
3.7.2. Método Matricial de las Rigideces 150
3.7.3. Extracción de Información para el Análisis Estructural . 156
3.7.4. MECA 1.0: Software para el Análisis Matricial de Rigideces 160

4. Resultados **167**
4.1. Medición de la Eficiencia y Flexibilidad del Modelo Paramétrico 167
4.2. Resultados en GiD 191
4.3. Modelos con Alto Número de Barras 198
4.4. Discusión 201

5. Conclusiones **203**
5.1. Diseño Paramétrico vs Proceso Tradicional 203
5.2. Trabajo Futuro 205

A. Implementación en Python **207**
A.1. Código: Torre Mayor 207
A.2. Código: Intersección de Niveles 212
A.3. Código: Retícula de Estructuración 214
A.4. Código: Obtención de Superficies a Partir de la Retícula 219
A.5. Códigos: Catálogos de Materiales 224
A.5.1. Código: Concreto 224
A.5.2. Código: Acero Rolado en Caliente 224
A.5.3. Código: Acero Rolado en Frío 224
A.6. Código: Generador Estructural 226
A.7. Código: Distribución de Cargas 237
A.8. Código: Análisis Estructural Utilizando MECA 246
A.9. Código: Resultados de Eficiencia Mecánica 250

B. *Structural Dynamo*:
Manual Básico de Usuario **252**
B.1. Instalación Local 252
B.2. Ejemplo Práctico: Análisis de una Edificación Rectangular . . . 254

C. Módulos de Visualización **271**
C.1. Visualizador de Estructuración 271
C.2. Visualizador de Distribución de Fuerzas 273
C.3. Visualizador de Comportamiento Estructural 276

Bibliografía **279**

Índice de figuras

1.1. Innovación en la Arquitectura. 6
1.2. Ingeniería Estructural. 8
1.3. Problemática en la Planeación. 11
1.4. Información en la Construcción. 12

2.1. Contexto Histórico. 21
2.2. Contexto Histórico. 21
2.3. Contexto Histórico. 23
2.4. Contexto Histórico. 23
2.5. Modelación tridimensional. 25
2.6. Curva de Paulson. 26
2.7. Curva de Macleamy. 27
2.8. Diseño Paramétrico en Arquitectura. 29
2.9. Håvard Vasshaug - Modelado Paramétrico. 31
2.10. Håvard Vasshaug - Modelado Paramétrico. 31
2.11. Håvard Vasshaug - Modelado de Edificaciones. 33
2.12. Håvard Vasshaug - Modelado de Edificaciones. 33
2.13. Diseño Paramétrico Computacional. 34
2.14. Curva de Boehm. 36
2.15. Curva de Boehm. 36
2.16. Data Flow Computing. 38
2.17. Múltiples diseños. 39
2.18. Data Flow Computing. 40
2.19. Edita-Compila-Corre. 42
2.20. Grulla Japonesa. 44
2.21. Programación Visual - Grulla Japonesa. 45
2.22. Complejidad Ciclomática. 51
2.23. Dynamo. 52
2.24. Dynamo. 54
2.25. Dynamo. 55
2.26. Dynamo. 57
2.27. Dynamo. 58
2.28. Dynamo. 58

2.29. Dynamo. 58
2.30. Dynamo. 60
2.31. Geometría en Dynamo. 61
2.32. Geometría en Dynamo. 61
2.33. Geometría en Dynamo. 63
2.34. Geometría en Dynamo. 63
2.35. Geometría en Dynamo. 63

3.1. Procedimiento para generar un DAG. 66
3.2. Procedimiento para generar un DAG. 66
3.3. Procedimiento para generar un DAG. 67
3.4. Procedimiento para generar un DAG. 68
3.5. Procedimiento para generar un DAG. 68
3.6. Procedimiento para generar un DAG. 69
3.7. Procedimiento para generar un DAG. 69
3.8. Procedimiento para generar un DAG. 70
3.9. Procedimiento para generar un DAG. 70
3.10. Procedimiento para generar un DAG. 71
3.11. Procedimiento para generar un DAG. 72
3.12. Procedimiento para generar un DAG. 72
3.13. Python Script. 75
3.14. Python Script. 76
3.15. Torre Mayor. 77
3.16. Factores de proporcionalidad. 77
3.17. Torre Mayor. 79
3.18. Torre Mayor. 80
3.19. Torre Mayor. 81
3.20. Torre Mayor. 82
3.21. Torre Mayor. 82
3.22. Torre Mayor. 83
3.23. Torre Mayor. 83
3.24. Torre Mayor. 84
3.25. Intersección de sólido. 86
3.26. Latencia de Intersección. 87
3.27. Programa Lineal. 88
3.28. Programa Paralelizado. 88
3.29. Aplicaciones de la Programación en Paralelo. 89
3.30. Paralelización de intersecciones. 91
3.31. Latencia de Intersección. 93
3.32. Curvas Perimetrales. 95
3.33. Curvas Perimetrales. 96
3.34. Estructuración. 97
3.35. Grid. 99
3.36. Flexibilidad de Estructuración. 99

3.37. Visualización de Grid. 100
3.38. Obtención de Superficies. 103
3.39. Paralelización de Intersecciones. 104
3.40. !Peligro! 108
3.41. Structure Generator. 112
3.42. Structure Generator. 113
3.43. Structure Generator. 113
3.44. Structure Generator. 114
3.45. Structure Generator. 114
3.46. Structure Generator. 114
3.47. Resultados de Structure Generator. 116
3.48. División de Curvas Perimetrales. 117
3.49. Metodología para estructuración externa. 118
3.50. Estructuración Externa 119
3.51. Estructuración Externa en Torre Mayor. 120
3.52. Inclinación en columnas. 121
3.53. Resultado de estructuración externa. 122
3.54. Resultado de estructuración externa e interna. 124
3.55. Problemática con vigas externas 126
3.56. ParameterAtPoint. 127
3.57. Ordenación de Puntos. 129
3.58. Resultado de Módulo ***Structure***. 132
3.59. Distribución de Fuerzas. 134
3.60. Métodos de Distribución de Cargas. 135
3.61. Distribución en vigas. 135
3.62. Distribución en vigas. 135
3.63. Distribución en vigas. 137
3.64. Métodos de Distribución de Cargas. 137
3.65. Métodos de Distribución de Cargas. 137
3.66. Métodos de Distribución de Cargas. 138
3.67. Métodos de Distribución de Cargas. 138
3.68. Métodos de Distribución de Cargas. 140
3.69. Métodos de Distribución de Cargas. 140
3.70. Referencia de nodos en losas. 142
3.71. Distribución de cargas en losas rectangulares. 143
3.72. Losas rectangulares. 145
3.73. Losas poligonales. 146
3.74. Búsqueda de vigas en losas. 147
3.75. Resultado de distribución de fuerzas. 148
3.76. El Tajín. 149
3.77. Fuerzas en Extremo Local. 151
3.78. Sistemas Local-Global. 153
3.79. Matriz de Transformación. 154
3.80. Fuerzas en Extremo Global. 156

3.81. Información Estructural. 158

3.82. MECA 1.0. 161

3.83. MECA 1.0. 162

3.84. MECA 1.0. 162

4.1. Complejidad Ciclomática. 170

4.2. **Latencia Edificio Rectangular.**Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "Rec Building", se observa que para modelos menores a 25 000 barras se necesita menos de 10 min para el proceso completo. Sin embargo para el modelo #4 de 37320 barras, el tiempo necesario es seis veces mayor, por lo que se presenta un comportamiento cuasi-exponencial. 174

4.3. **Latencia Edificio Elipsoidal.**Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "Ellipse". En este caso se ejecutaron modelos más pequeños, por lo que se observa un buen comportamiento, donde mientras crece el número de barras también lo hace el tiempo de forma regular. Aquí vemos la mayoría de los modelos están por debajo de los dos minutos, lo cual es algo muy bueno ya que son edificios de tamaño regular que se construyen de manera más común que rascacielos. 178

4.4. **Latencia Edificio Elipsoidal.**Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "3Pol Building". Este es un caso interesante, primero porque los modelos son pequeños, se trató de ejecutar análisis de edificaciones que podemos encontrar casi en cualquier ciudad, con no más de 12 000 barras. Por tanto los tiempos de ejecución son menores a un minuto en su mayoría. Una particularidad de esta gráfica es que al inicio de las curvas se tiene un crecimiento de latencia y luego disminución. Esto ocurre debido a que inicialmente el algoritmo no tiene activada la función de paralelización por las razones que se dieron en el capítulo correspondiente, es por ello que al activarse automáticamente, se tiene una reducción importante de tiempo en el cuarto punto mostrado. 182

4.5. **Latencia Torre Mayor.**Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación "Torre Mayor". Este es el caso de estudio principal que se tuvo en este trabajo. Algunas cuestiones interesantes son por ejemplo que el modelo #1 se refiere a las dimensiones y estructuración mas cercana a la realidad en base a los bocetos encontrados de la Torre Mayor. En cuanto a la latencia, ocurre un comportamiento similar al modelo rectangular. Se tiene la mayor parte de los modelos alrededor de los diez minutos, sin embargo el modelo de 43 846 Barras se dispara a 45 min, esto es 3 veces más que el modelo anterior que tiene solo la mitad de barras. Además se observa un pico en la gráfica, a pesar de distintas pruebas sigue ocurriendo este comportamiento, y se podría deber a que este modelo en particular necesita de una búsqueda más robusta de elementos exteriores, por lo que retrasa tanto la generación de la estructura como la distribución de losas. 186
4.6. **Comparación de Latencia Torre Mayor.**Gráfica con escala lineal obtenida a partir de la comparación de resultados en el modelo de la"Torre Mayor" al generarse en dos equipos de cómputo distintos. Los modelos que se generaron con el equipo de supercómputo se presentan en color anaranjado mientras que los generados por un equipo de cómputo común tienen color verde 190
4.7. Momentos Internos - GiD. 192
4.8. Reducción de Desplazamientos - GiD. 193
4.9. Reducción de Eficiencias - GiD. 194
4.10. Asignación de Secciones/Materiales y Cargas - GiD. 195
4.11. Análisis de Momentos y Eficiencias - GiD. 196
4.12. Desplazamientos - GiD. 197
4.13. Edificación Pentagonal. 199
4.14. Edificación Triangular. 200

5.1. Diseño Paramétrico vs Proceso Tradicional. 204
5.2. Optimización Estructural. 205

B.1. Verificación de Instalación. 253
B.2. Ventana Principal Dynamo. 254
B.3. **Biblioteca Dynamo.** Opciones para generar un programa visual en **Dynamo**. Se muestra el contenido del paquete ***Structural Dynamo***. 255
B.4. **Sólido Tridimensional.** Selección de modelo de edificación rectangular. 256

B.5. **Sólido Tridimensional.** Parámetros para generar el sólido tridimensional. . . . 256
B.6. **Sólido Tridimensional.** Creación de grupos por color. . . . 257
B.7. **Sólido Tridimensional.** Generación de sólido al ejecutar el programa visual. . . . 258
B.8. **Sólido Tridimensional.** Opción "Automatic" para generación rápida de modelos. . . . 259
B.9. **Sólido Tridimensional.** Sólido a estructurar. . . . 259
B.10. **Retícula de Estructuración.** Se añade el nodo ***Grid*** para proponer una estructuración. . . . 260
B.11. **Retícula de Estructuración.** Parámetros de estructuración. . 260
B.12. **Retícula de Estructuración.** Visualización de ***Grid*** . . . 261
B.13. **Retícula de Estructuración.** Visualización de ***Grid*** . . . 261
B.14. **Generación de Estructura.** Se añade el nodo ***Structure***. . . 262
B.15. **Generación de Estructura.** Material y sección del catálogo de **"MECA"**. . . . 262
B.16. **Generación de Estructura.** Conexión de materiales y visualización de elementos estructurales. . . . 263
B.17. **Generación de Estructura.** Visualización de elementos estructurales. . . . 264
B.18. **Distribución de Cargas.** Se añade el módulo ***SlabLoads***. . . 264
B.19. **Distribución de Cargas.** Se muestra la conexión correcta para la asignación de cargas mediante ***SlabLoads***. . . . 265
B.20. **Distribución de Cargas.** Resultado del módulo ***SlabLoads***. . 266
B.21. **Análisis Estructural.** Se añade y se conecta el módulo ***MECA_StructuralAnalysis***. . . . 266
B.22. **Análisis Estructural.** Incorporación y cambio de nombre de Boolean para ***MECA_StructuralAnalysis***. . . . 267
B.23. **Análisis Estructural.** Conexión correcta para ***MECA_StructuralAnalysis***. 268
B.24. **Análisis Estructural.** Módulo de visualización y resultado de eficiencias para ***MECA_StructuralAnalysis***. . . . 268
B.25. **Modificación de Parámetros Estructurales.** Al tener conectado el programa visual, es muy sencillo modificar valores en los parámetros para obtener una nueva propuesta estructural. . 269
B.26. **Modificación de Parámetros en Sólido.** Si se modifican las dimensiones del sólido tridimensional, la estructuración seguirá funcionando de la misma manera, por lo que se pueden generar más soluciones de acuerdo a los cambios que sufre un edificio.. . 270

C.1. Visualización Grid. . . . 272
C.2. Visualización Grid. . . . 272
C.3. Visualización Structure. . . . 273
C.4. Visualización Structure. . . . 274
C.5. Visualización Slab Loads. . . . 274

C.6. Visualización Slab Loads. 275
C.7. Visualización Eficiencias. 277
C.8. Visualización Eficiencias. 278

Índice de tablas

2.1. Resultados del análisis realizado por Davis (9), se muestran los diez nodos más comúnes en los 2002 modelos verificados. Estos nodos representan el 40 % de los 93 530 nodos existentes. 48

3.3. Tabla de relación de extensiones de archivos utilizados por el software **MECA**.161
3.4. Tabla de recomendaciones de selección de método numérico para solución de sistemas de ecuaciones por el software **MECA**, se presenta de forma que sirva como reseña histórica ya que la memoria utilizada por las computadoras de hoy es por mucho más grande. 163

4.6. **Líneas de código LOC.**Tabla que muestra la comparación entre la medición de líneas de código para modelos paramétricos usando redes DAG y el número de líneas programadas con lenguaje Python para cada módulo propuesto en este trabajo. 169
4.7. Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de un prisma rectangular. Se incluyen los tiempos de ejecución o latencia por cada modelo. 175
4.8. Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de un prisma elipsoidal. Se incluyen los tiempos de ejecución o latencia por cada modelo. 179
4.9. Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de varios prismas generados por el módulo "3Pol". Se incluyen los tiempos de ejecución o latencia por cada modelo. . . . 183
4.10. Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de la "Torre Mayor". Se incluyen los tiempos de ejecución o latencia por cada modelo. 187

C.15. Tabla de relación de colores presentados por *Postproceso* en el módulo de ***Analysis*** programado en **Dynamo**. 277

Oculus: Arquitectura e Ingeniería Moderna. Estación de tren localizada en Nueva York, obra del arquitecto e Ingeniero Santiago Calatrava. Inspiración para la realización de diseños complejos e innovadores. (E. Muttio - 2016)

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, el área de la construcción en México ha evolucionado de manera gradual junto con las herramientas tecnológicas que surgen cada día. Las propuestas arquitectónicas nacionales compiten con el ámbito internacional, los empresarios invierten en arquitectura innovadora y compleja, que claramente marcan un reto para los proyectistas. Entre éstos se encuentran los Ingenieros Estructurales, quienes son los encargados de hacer una propuesta que dará el soporte necesario a la obra arquitectónica, brindando la alternativa económica ideal y comprometiéndose en la seguridad de la edificación.

Algunos ejemplos en nuestro país en los que el diseño en edificaciones ha tenido un desarrollo innovador mediante propuestas complejas son: el Museo Soumaya, localizado en la Ciudad de México, el cual tiene un diseño "orgánico" formado por una fachada compuesta por hexágonos y cuya estructura tiene una configuración muy especial, lo cual resulta en un diseño arquitectónico y estructural interesante (fig. 1.1 a)); el proyecto de un supermercado en Santa Fé (CDMX), que tiene una planeación basada en un software de *Modelado de Información de la Construcción* o *BIM*, que permite una mejor gestión de la logística y por ello otorga la posibilidad de crear complejos diseños arquitectónicos como el mostrado en la imagen 1.1 b); otro ejemplo es la fachada de la tienda departamental en Insurgentes (CDMX), la cual está compuesta por paneles de vidrio con diversas formas geométricas por lo que da un aspecto estético , siendo éste un desafío en el proyecto y construcción (fig. 1.1 c)); y por último, la Cineteca Nacional Siglo XXI (CDMX) muestra un diseño arquitectónico basado en una plataforma con repetición de geometrías, la distribución de espacios y la gestión de los componentes es muy complicada por lo que el uso de recursos tecnológicos se hace necesario para desarrollar este proyecto, como se puede observar en la imagen 1.1 d).

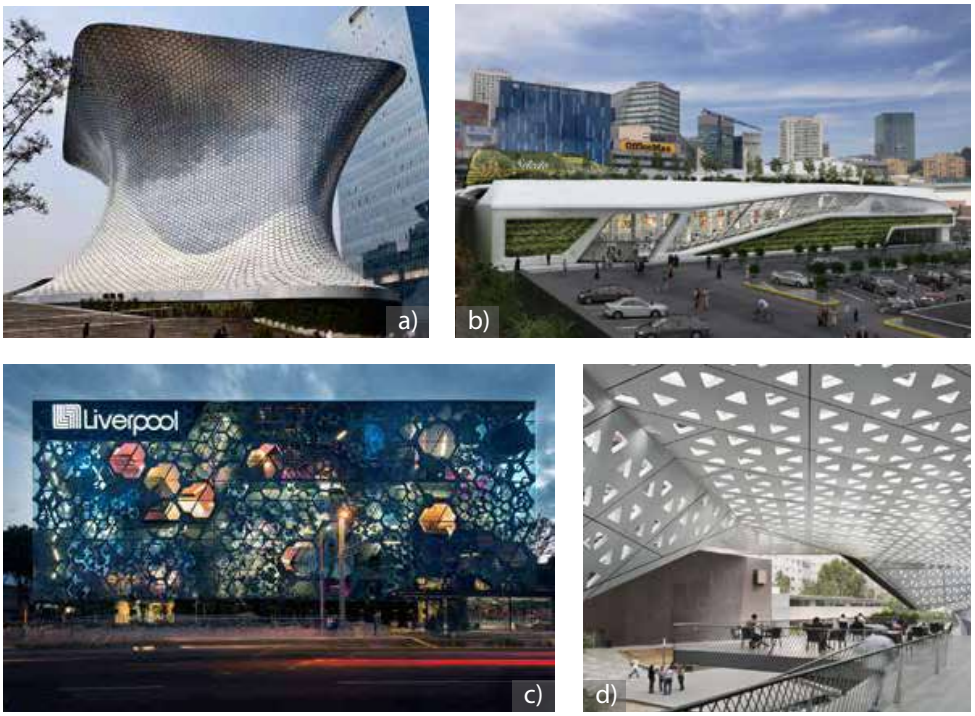


Figura 1.2: Innovación en la Arquitectura. Ejemplos de edificaciones que muestran un avance importante en la manera de hacer arquitectura de nuestro país, los proyectos son cada vez más complejos de analizar. ((30), (53), (3), (2))

Últimamente se han lanzado al mercado herramientas que ayudan a realizar diseños arquitectónicos con más facilidad, dando libertad a los arquitectos de proponer modelos más complejos e interesantes. De igual manera, para poder diseñar estructuras que satisfagan dichas necesidades arquitectónicas con alto grado de complejidad como las mostradas, se ha desarrollado software relacionado con la Ingeniería Estructural y se puede decir que en estos tiempos, ya existe una alta diversidad de programas computacionales especializados en el cálculo numérico de estructuras. Sin embargo, dado que los proyectos son más complejos cada vez sobre todo en cuestión de la logística que debe seguir un equipo de proyectistas (tanto arquitectos como ingenieros), se requieren nuevos procedimientos y herramientas que nos ayuden a resolver problemas que trabajen en conjunto con los programas de cálculo estructural. Es de esta manera que se puede pensar en una metodología multidisciplinaria que permita una colaboración más eficiente entre personas que trabajan en distintos componentes del mismo proyecto.

1.1.1. Procedimiento Tradicional en la Ingeniería Estructural

La Ingeniería Civil tiene una fuerte conexión con los desarrollos científicos y tecnológicos de la historia. Desde los primeros avances en la Mecánica de Materiales en los siglos XVII y XVIII, hasta los avances en computación de las últimas décadas se ha observado que cada vez los proyectos exigen una ca-

pacidad de cálculo mayor. Y si consideramos que una simple estructura puede producir miles de incógnitas a evaluar en aproximaciones matriciales, resulta que el análisis numérico es casi imposible sin la ayuda de una computadora (57).

En la mayoría de proyectos relacionados con la Ingenieria Civil es posible separar los procedimientos necesarios para su realización, tal como lo podemos observar en la figura 1.2. En este diagrama, se planteó una separación de procesos en cuatro módulos importantes. La primer etapa denominada **Datos de Entrada** es donde el ingeniero requiere hacer una interpretación a partir de planos arquitectónicos para obtener información en cuanto a la geometría y disposición de elementos portantes; posteriormente debe proponer un modelo adecuado para la estructura, y se tomarán decisiones como el tipo de estructuración, sistemas constructivos, entre otros. Aquí también se proponen materiales a utilizar, de acuerdo al tipo de proyecto y disponibilidad de éstos en la zona, además se deben tener en cuenta sus correspondientes propiedades mecánicas a incluir en el modelo. Dependiendo el proyecto se deben estimar las fuerzas que intervendrán, tanto gravitacionales por uso del edificio como fuerzas debido a otros factores como pueden ser sismos o vientos.

El siguiente módulo es llamado **Análisis Estructural**, en donde se debe seleccionar el tipo de análisis a realizar, el cual es una parte fundamental ya que debe ser congruente con el modelo propuesto antes, como se verá posteriormente el método más empleado por los programas comerciales es el denominado *Método Matricial de las Rigideces* y aunque no es el único existente, es adecuado utilizarlo para estructuras con múltiples barras. Durante esta etapa se debe verificar los datos a ingresar en el software, este procedimiento requiere de una cantidad importante de tiempo, y se realiza justo antes del cálculo numérico. Será necesario que el usuario utilice otros medios auxiliares además del software de análisis estructural para gestionar los datos de entrada, como lo son las hojas de cálculo y plataformas CAD. Posterior a la incorporación de dichos datos en el software de análisis, se realiza el cálculo de la estructura de acuerdo al modelo propuesto y al tipo de problema para conocer el comportamiento de la misma en base a dos resultados principales, los cuales son los desplazamientos nodales y las fuerzas internas de barras.

La etapa de **Resultados**, proveniente del cálculo numérico que el software de análisis estructural entrega, se puede considerar como la información de salida que describe el comportamiento del sistema estructural, y estos datos serán necesarios para comenzar otro proceso denominado **Diseño Estructural**, en el cual el ingeniero civil debe verificar los resultados obtenidos usando un "código de construcción", que le ayudará a decidir si la propuesta es adecuada en base a la resistencia del material, en éste se incluyen las dimensiones de las secciones transversales, los materiales empleados, la estructuración, entre otros. Si la verificación de la resistencia tiene una respuesta positiva, entonces

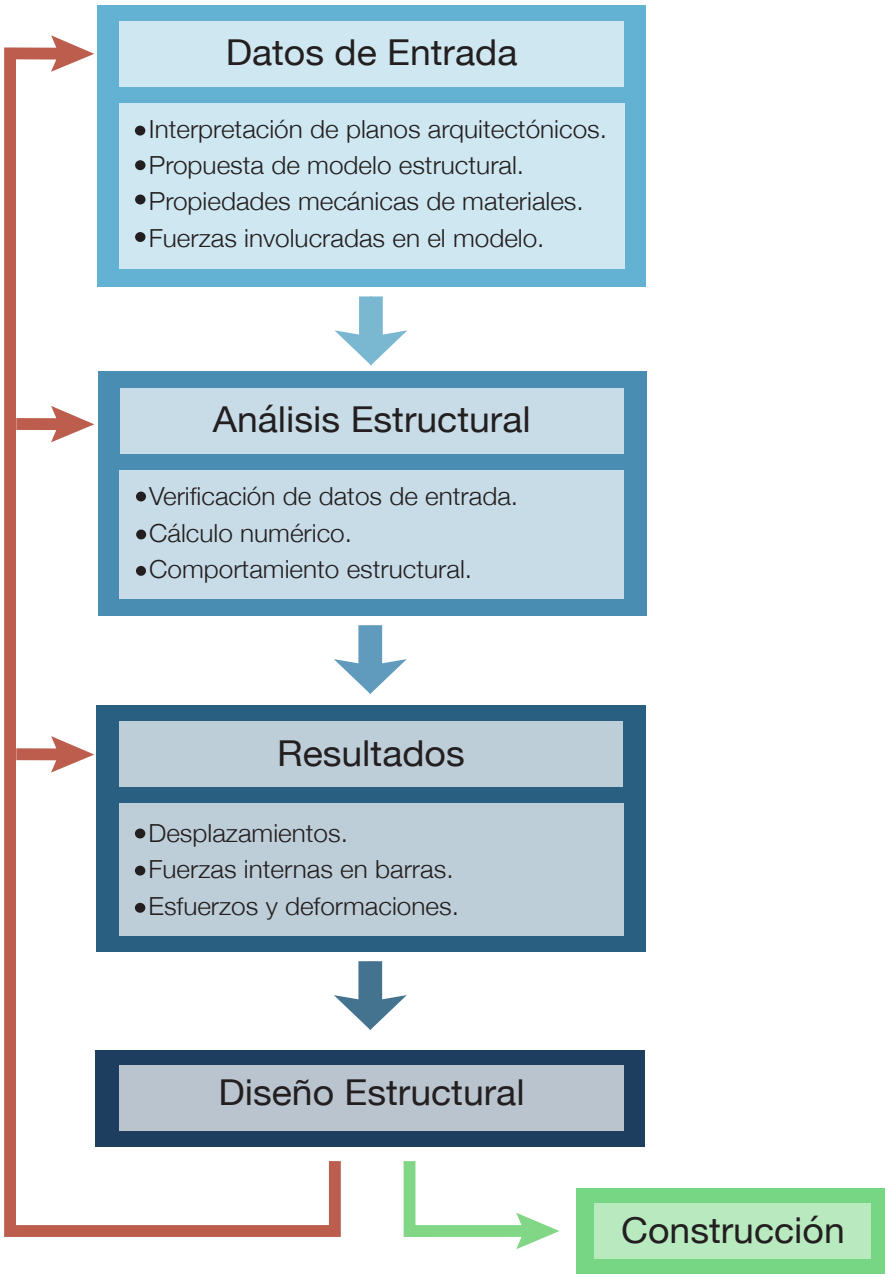


Figura 1.3: Ingeniería Estructural. Diagrama del procedimiento tradicional realizado durante un proyecto de análisis y diseño estructural.

se puede plantear continuar con el proceso del proyecto, que sería la elaboración de planos con la información estructural para comenzar a construir la edificación, tal como se muestra en la imagen con una flecha color verde. Pero si durante el proceso de cálculo se tiene una respuesta negativa aún cuando se están utilizando factores de seguridad, entonces hay que regresar a una parte del proyecto y modificar algún parámetro. Éste puede re-plantearse desde la primer etapa del ingreso de datos o posiblemente durante la elección del modelo estructural, ya que quizá no haya sido el conveniente, o los materiales de las secciones transversales eran insuficientes. Pudieran ser diversas circunstancias, por ello estos cálculos al final del proceso darán una pista para saber qué parámetro es conveniente cambiar y no tener que modificar en gran medida la propuesta realizada.

Por otra parte, el tiempo requerido para un diseño estructural no está definido, sino que éste dependerá de la complejidad del proyecto. No obstante, por pequeño que éste sea, la obtención de datos y el análisis de la estructura pueden requerir semanas. Aún cuando existe un mayor desarrollo de herramientas, se ha visto que hay un cierto aislamiento en estos programas, es decir, la obtención de datos de entrada, el cálculo numérico y el análisis del resultado están separados unos de los otros. Por ejemplo, si se desea analizar un sistema estructural, se deberán ingresar datos de entrada en el software de análisis, y además se deberá hacer una verificación de estos datos, ya que se debe cuidar que no se ingrese información equivocada, inclusive cuando la información sea correcta y verificada en alguna hoja de cálculo o en un plano. Estos procesos de múltiples verificaciones o de corrección de datos demoran cierto tiempo, por lo que es de especial interés reducir en lo posible estos retrasos, sobretodo en la práctica profesional cuando el tiempo es un recurso invaluable.

1.1.2. Principales Problemas en el Proceso

Como explicación de las problemáticas encontradas durante el proceso que conlleva un proyecto constructivo, tomemos como ejemplo el siguiente supuesto que comúnmente ocurre en los proyectos de ingeniería y que está representado en el grupo de imágenes de la figura 1.3. Se desea construir un edificio de tipo residencial, para el cual se realiza un diseño arquitectónico. Para transmitir la información de éste es necesario elaborar planos arquitectónicos, que aunque no tienen un grado alto de detalle, se invierte cierta cantidad de tiempo tal como se observa en la figura 1.3 a). Posteriormente estos planos se proporcionan a una oficina de Ingeniería, donde se procede con los pasos para la estructuración y su análisis explicados en la sección 1.1.1 y en el diagrama de la figura 1.2; la figura 1.3 b) representa la configuración estructural propuesta. Después de algunos días de trabajo, se pide hacer un cambio importante en el diseño arquitectónico inicial, en este caso la edificación tiene una dimensión

superior (figura 1.3 b)), que repercutirá en una modificación en los elementos portantes, los cuales serán mas largos (figura 1.3 d)).

Considerando el ejemplo anterior, **¿El diseño estructural será el mismo?** Es una pregunta que solo podría contestarse acertadamente revisando a detalle el proceso de diseño, ya que hay cierta posibilidad de que éste siga siendo útil, pero el Ingeniero Civil no puede darse el lujo de "confiar" en que no se afectará la resistencia de los elementos. Se deben considerar los cambios realizados, lo que repercute en volver al inicio del proceso de análisis y diseño estructural.

En la práctica profesional, las oficinas de Ingeniería afrontan modificaciones constantes en los proyectos, desde cuestiones arquitectónicas, cambios en los materiales o alteraciones en el procedimiento constructivo, por lo que hay que realizar iteraciones en puntos clave del proceso. Aunque el ejemplo mostrado en la figura 1.3 es muy simple o quizá "burdo", la mayoría de las ocasiones ocurren estas situaciones por lo que las oficinas de arquitectura tienen que entablar mucha retroalimentación con las oficinas de ingeniería (en el caso de estar separadas), lo que conlleva un ciclo repetitivo hasta alcanzar el diseño definitivo. Otros ejemplos en este proceso, son el desplazamiento de elementos estructurales debido a que se pueden colocar elementos arquitectónicos como ventanas, por lo que modifica la configuración estructural; otro más sería modificación de secciones transversales de vigas, ya que cuando éstas son muy altas se pueden considerar poco estéticas y es recomendable buscar otra solución, como el cambio de material de concreto a acero; un ejemplo más pudiera ser la consideración de varias propuestas arquitectónicas que pueden ser muy distintas y aunque de manera conceptual se pide proponer el sistema estructural para determinar un costo aproximado, se necesita varios modelos que pueden ser elegidos para continuar con el proceso.

Un esquema con una descripción más detallada se basa en el diagrama de Sacks y Barak (40) mostrado en la figura 1.4, donde se detallan algunos pasos que se deben seguir para completar un proyecto de construcción. Hágase notar que se incluyen los ciclos donde existe una retroalimentación o "feedback" entre varios participantes y además hay puntos donde no se puede continuar hasta tener un visto bueno o un acuerdo en dicho procedimiento, lo que ralentiza el proceso completo. Justo iniciando el diagrama, se observa la fase de **Pre-diseño**, en la que algunas veces los ingenieros tienen cierta información disponible desde el principio, pero algunas ocasiones hay que esperar al diseño arquitectónico conceptual. Una vez que se ha hecho el cálculo se tiene un ciclo de retroalimentación y sucede lo comentado antes con respecto a la decisión del modelo estructural.

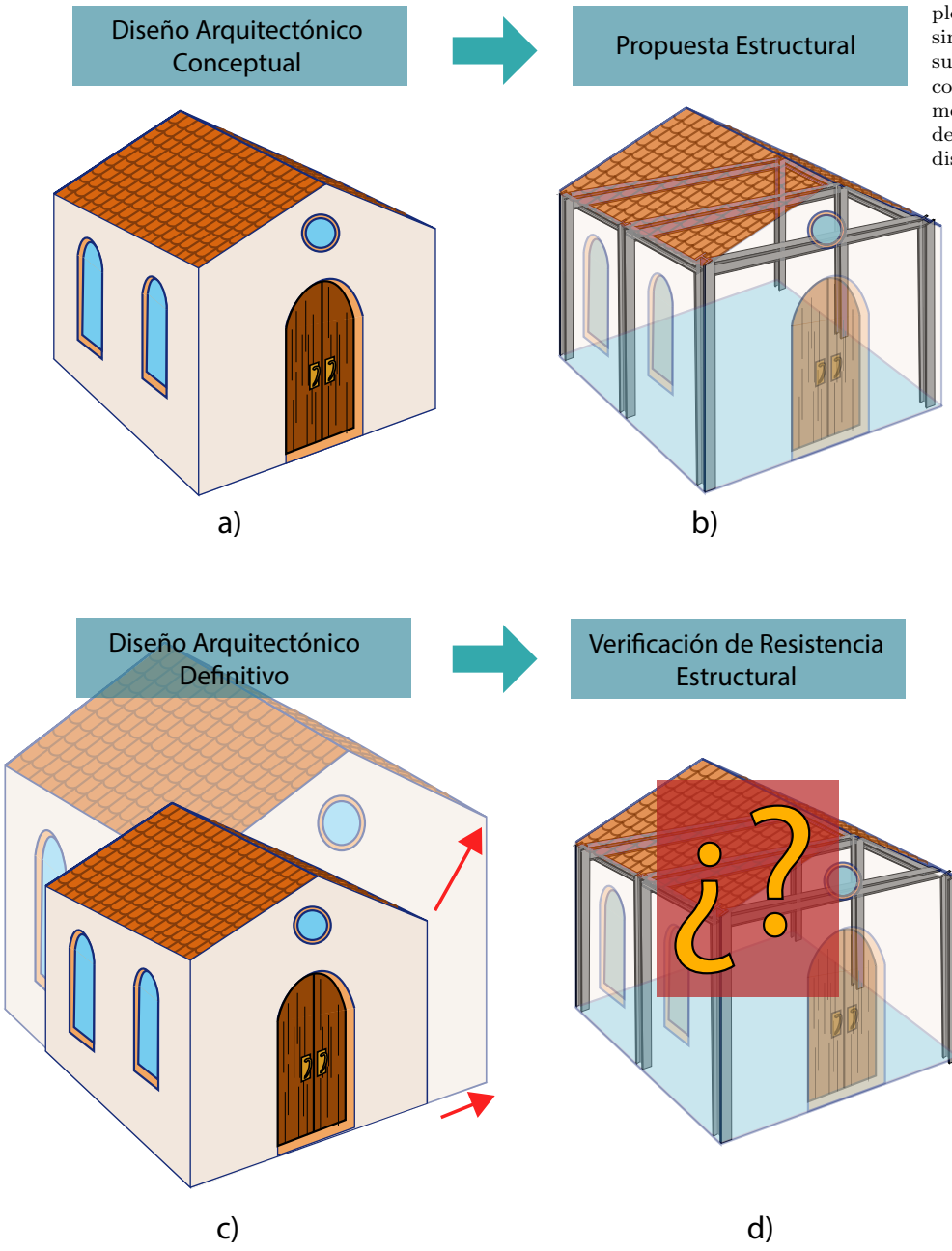


Figura 1.4: Problemática en la Planeación. Ejemplo que retrata de manera simple los cambios que puede sufrir cualquier proyecto de construcción. En este caso la modificación de dimensiones de una casa y su afectación al diseño estructural.

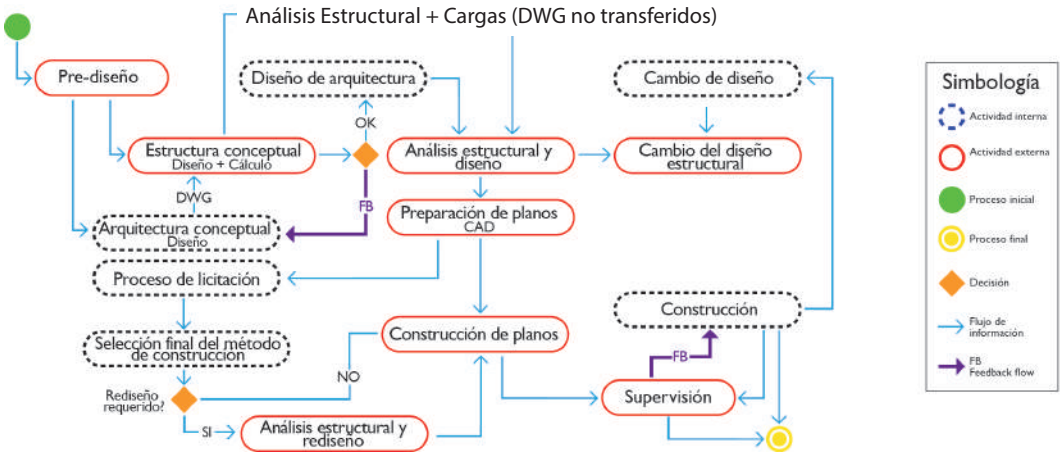


Figura 1.5: Información en la Construcción. Esquema general del proceso de diseño conceptual arquitectónico y el análisis estructural que hacen los ingenieros (40).

Cuando el diseño conceptual llega a un acuerdo, ahora si se hace un detallado de los componentes y se revisa con cuidado los cálculos para posteriormente trazar mediante un programa CAD comercial los planos para comenzar un proceso de *licitación*, en el cual se decide por el proyecto más adecuado. Al momento de seleccionar la propuesta, algunas veces se pide hacer re-diseños, que conllevan otro ciclo de análisis y verificación. Terminando esto, se comienzan a trazar los planos definitivos marcando detalles acerca del proceso constructivo, para que posterior a una supervisión, se pueda dar inicio la construcción del proyecto, aquí también existe mucha retroalimentación entre constructores y supervisores. Y por si fuera poco, también pueden ocurrir problemas en la propuesta, por lo que algunas veces se tiene que plantear un cambio de diseño y hay que regresar hasta un punto que se le pueda dar solución a ese problema.

Un aspecto importante de este diagrama es que se marcan algunos pasos donde se utilizan archivos *DWG*, los cuales son empleados por el programa de diseño asistido por computadora CAD comercial más utilizado por la industria de la construcción. Entonces la importancia de mostrar esto es que el recurso para transmisión de información es utilizando estos archivos, y como podemos ver existen muchos ciclos donde se deben modificar. Por lo que los avances de la computación ayudaron a que se pudiera pasar del trazo de planos directamente en papel a un sistema computacional más sencillo de utilizar, pero ahora la logística de proyectos complejos requiere nuevas herramientas que permitan hacer un trabajo más eficiente y reducir el tiempo que se tardan en estos pasos.

1.1.3. Solución Propuesta

En base a lo mencionado en la sección anterior, sabemos que los procesos en ingeniería avanzan en conjunto con la innovación en la tecnología. Es por ello

que cuando el diseño asistido por computadora CAD apareció comercialmente, tuvo un impacto muy grande en la eficiencia de los proyectos. Pero ahora, los proyectos son más ambiciosos ya que la arquitectura se vuelve más compleja, de la misma manera la Ingeniería Estructural y la logística que conlleva realizar un proyecto. Y como se analizó mediante los diagramas de las figuras 1.2 y 1.4, existen muchos ciclos que se necesitan realizar cada vez que existe una retroalimentación o modificación de componentes. Entonces el costo de esfuerzo y tiempo en la planeación y ejecución de un proyecto se ve afectado por estos cambios. También se mencionó que la información utilizada en la industria de la construcción esta basada en planos trazados con los sistemas CAD, los cuales tienen la desventaja de que cada vez que existen modificaciones tienen que actualizarse, e incluso algunas ocasiones borrar componentes para ser reemplazados por otros. La idea es buscar una alternativa en la gestión de la información para que la actualización de información no demore tanto. En caso específico, manejar con mayor eficiencia la información estructural.

Aunque la idea no es reemplazar el dibujo sobre planos bidimensionales, se propone utilizar un modelo digital que contenga la información de todos sus componentes y también información relevante para realizar un análisis estructural y posteriormente revisar su resistencia. Al utilizar un modelo de esta índole, es más sencillo modificar los datos que contiene para proponer distintas soluciones. Estos datos son conocidos como *parámetros*, y se están desarrollando herramientas basadas en parametrización geométrica y de procesos para reducir el costo de los cambios. El diseño paramétrico ofrece alternativas en cuanto al diseño y optimización de proyectos. Su planteamiento está basado en relaciones matemáticas e interdependencias de parámetros, generando múltiples soluciones y disminuyendo el tiempo en la realización de modificaciones. Para poder implementar una metodología en base a parámetros, es necesario conocer todas la variables que son partícipes en un proyecto estructural, esto lo vuelve muy complicado porque cada proyecto es distinto. Por ello se reducirá el campo de aplicación a edificaciones tipo torre y estructuración en base a marcos rígidos o pórticos. Es necesario poseer una habilidad especial para saber restringir la forma, el tamaño, la composición, la localización, la orientación de piezas y conexiones porque de esta manera los diseñadores pueden incorporar modelos de edificios a un entorno paramétrico.

Diversas estructuras, como las edificaciones, están compuestas por múltiples partes que interactúan juntas, todas organizadas en varias secciones y las cuales deben de ser planeadas de manera integral, sobretodo si se desea tener un modelo que describa el comportamiento del conjunto. Una ventaja que podemos encontrar, es que debido a la estandarización (que ayuda a mantener una mejor logística), muchas de las partes son idénticas y usadas repetidamente, solo algunas variaciones entre etapas de la construcción se hacen comúnes. Es por ello que la incorporación de modelos de edificaciones en una plataforma

con enfoque al diseño paramétrico, puede garantizar una ventaja durante la planeación de construcciones (41). Sacks y Barak (2007) (40) describen algunas de las ventajas de usar el modelado paramétrico en proyectos de edificaciones:

- Los modelos tienen la capacidad de incorporar información estructural y de construcción.
- Se puede hacer un análisis dentro del modelador sin necesidad de extracción y re-entrada de datos en otro software, ya que el modelo de análisis puede ser derivado del modelo paramétrico.
- Algunos modeladores pemiten la interacción directa con software de dibujo de planos CAD, con poca interacción del usuario. Por lo que no se requiere empezar un dibujo desde cero en un software CAD.
- Los errores de escritura de datos se disminuyen. Esto debido a que los datos se introducen una sola vez en el modelo y éste se encarga de enviarlos a los programas vinculados.
- A diferencia de un plano CAD, un modelo paramétrico distingue los cambios realizados en algún punto del proceso y es ajustado automáticamente.
- Un modelo tridimensional da la posibilidad a los consultores de verificar un modelo 3D, ahorrando el costo y tiempo de realización de documentos y planos.
- Los procesos automáticos de optimización son sencillos de implementar y la búsqueda de soluciones se hace más eficientemente.

Un sistema de modelado paramétrico provee todas las funcionalidades necesitadas para rápidamente definir un diseño de edificación. Un usuario puede establecer una retícula, seleccionar e insertar elementos del edificio relativos a la retícula, incluyendo tamaño, las piezas, conexiones, etc. El modelo puede ser ajustado cambiando los valores de los parámetros en cada nivel, cualquier adaptación geométrica requiere mantener la integridad que es manejada por el sistema. Esto facilita la exploración de las alternativas de diseño, potencialmente aumentando la calidad del diseño resultante. Sin embargo, mucha más automatización es posible. A continuación se describen tres niveles a los que se les podría implementar una parametrización en sus procedimientos (40):

Análisis Básicos de Ingeniería: Si las cargas y las condiciones de carga son aplicadas directamente a un modelo de edificio, la entrada para el análisis estructural (u otros) pueden ser preparadas y transferidas automáticamente. Las reacciones resultantes pueden ser mandadas de nuevo al modelo paramétrico, donde pueden formar las bases para el refinamiento automatizado de los

parámetros de diseño de piezas individuales y de conexión. Esto reduce considerablemente la carga de trabajo de ingeniería requerida.

Diseño de elementos Estructurales y Conexiones: Existe una alta variedad de tipos de análisis y métodos de cálculo especializados en el diseño de componentes de construcción, como lo son las conexiones de los elementos estructurales. Éstos se implementan en los códigos típicos de regulaciones de ingeniería y manuales de diseño, e incluso módulos de software tienen disponible estos procesos. Estas aplicaciones pueden ser fácilmente ligadas para ser leídas y escritas como valores parámetro de un objeto, permitiendo que las piezas verifiquen su diseño automáticamente en base a los resultados obtenidos de un análisis.

Optimización del Diseño en Ingeniería: Para un trabajo de diseño que sea repetitivo y manejado por reglas, como por ejemplo un sistema de piso, un núcleo de la edificación, o incluso en algunos casos el edificio entero, el modelo puede ser planteado mediante simples parámetros. Un diseño de alto nivel automatizado y el análisis de esta naturaleza ha sido un objetivo de la investigación en el área de la construcción mediante la integración de la computación. El desarrollo reciente de una plataforma de modelado 3D completamente paramétrica y el incremento en el desempeño del hardware reciente puede abrir camino a la implementación de algoritmos que de manera automática verifiquen y propongan un diseño en base a una solicitación, lo que podríamos llamar como optimización en ingeniería utilizando la parametrización de sus procesos.

Por lo que una propuesta de solución al problema descrito es hacer una revisión del material disponible sobre esta metodología denominada *"Diseño Paramétrico"*, que permita localizar posibles beneficios en el área de la Ingeniería Estructural. Teniendo como fundamento estos conceptos, la idea es desarrollar una aplicación computacional que ayude a disminuir el tiempo requerido para el análisis y diseño de edificios.

1.2. Objetivo

El presente proyecto pretende contribuir con avances en temas relacionados a la Ingeniería Estructural de edificaciones utilizando herramientas computacionales. Se planea abordar el área que abarca la metodología conocida como *Diseño Paramétrico*, desde el contexto histórico, la planeación temprana basada en la utilización de parámetros iniciales, su relación con los sistemas computacionales y la modelación de sólidos, así como las aplicaciones en Arquitectura e Ingeniería. En base a esto, se pretende explicar el desarrollo de una aplicación, cuyo propósito esta centrado en la estructuración automática de edificaciones en base a su geometría tridimensional, así como la obtención de los datos necesarios para que un software de análisis estructural basado en el Método Matricial de las Rigideces realice el cálculo numérico y regrese los resultados del análisis. En base a la parametrización se pueden obtener múltiples soluciones para un problema estructural y así se pueda tener un enfoque nuevo en el comportamiento de las estructuras y la configuración de éstas en las edificaciones. El proyecto se planea programar en un lenguaje de alto nivel, donde se modelen diversas geometrías parametrizables simulando edificios, incorporando un entorno visual para poder observar el comportamiento del algoritmo en cuanto a su estructuración. Se podría considerar como el desarrollo de un módulo que permita ingresar datos de manera sencilla a un software de análisis, permitiendo una reducción de tiempo considerable debido a sus propiedades paramétricas y el buen uso de la información, su definición podría semejarse a un *"Pre-Proceso Dinámico"*.

1.2.1. Objetivos Específicos

1. Realizar una investigación exhaustiva acerca del *Diseño Paramétrico*, enfatizando en la aplicación de la Ingeniería Estructural.

2. Desarrollar una metodología más simple en base a la programación de una herramienta computacional para proponer una solución estructural de una edificación.

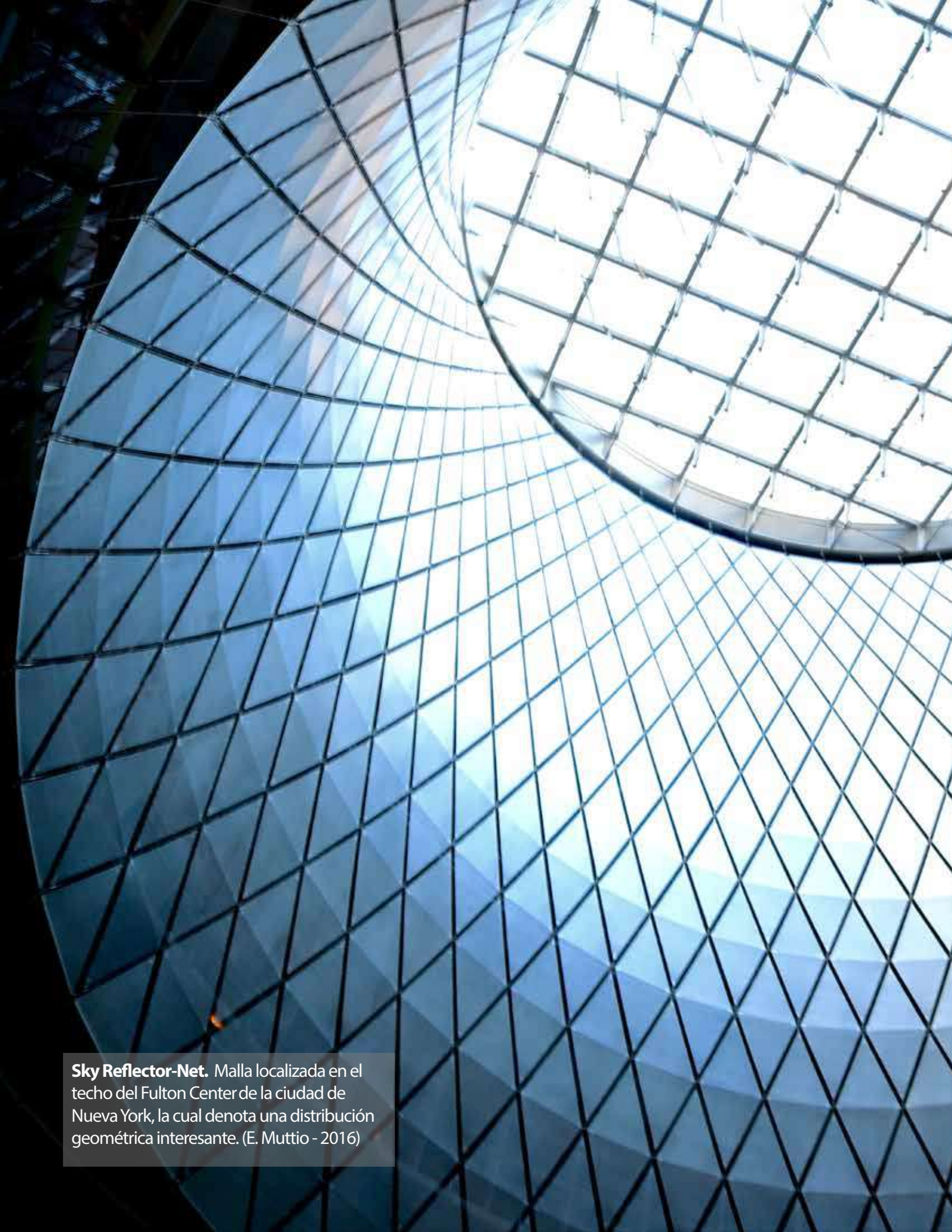
3. Parametrizar entidades, empleando librerías relacionadas con la generación de geometrías, que funcionen como modelos tridimensionales de edificaciones tipo torre utilizando el paradigma de programación visual.

4. Programar funciones, módulos u objetos utilizando un lenguaje de programación de alto nivel, que contengan algoritmos capaces de generar y utilizar información estructural a partir de geometrías parametrizables.

5. Incorporar módulos de solicitaciones o cargas gravitacionales que serán impuestas a las edificaciones.

6. Exportar información necesaria para realizar un análisis y diseño estructural, utilizando un software basado en el Método Matricial de las Rigideces.

7. Importar los resultados del software de análisis, para observer el comportamiento estructural dentro del ambiente de modelado paramétrico.



Sky Reflector-Net. Malla localizada en el techo del Fulton Center de la ciudad de Nueva York, la cual denota una distribución geométrica interesante. (E. Muttio - 2016)

Capítulo 2

Parametrización Geométrica

La tendencia actual en la arquitectura es la búsqueda de espacios eficientes mediante un diseño innovador y con mínimos recursos energéticos. Debido a esto, la complejidad de los proyectos y los trabajos realizados por los ingenieros civiles crece exponencialmente. Se ha observado que la logística de proyectos se vuelve cada vez más difícil, por ello el tiempo empleado para cada etapa de planeación debe ser óptima. La interpretación de los planos arquitectónicos para proponer una configuración estructural es un problema de conceptualización geométrica. Además de esto, la propuesta de soluciones estructurales requieren de un proceso de cálculo numérico que puede ser costoso computacionalmente, dependiendo de la complejidad de la estructura. En la práctica profesional, las oficinas de Ingeniería afrontan modificaciones constantes en los proyectos, desde cuestiones arquitectónicas, cambios en los materiales o alteraciones en el procedimiento constructivo, por lo que hay que realizar iteraciones en puntos clave del proceso. La parametrización de procesos, ayuda a reducir los ciclos debido a los cambios que ocurren durante la ejecución de los trabajos, lo que repercutirá en un ahorro de tiempo y un óptimo control del proyecto.

2.1. Primeros Avances en la Modelación Geométrica Computacional

El diseño en cualquier campo de la Ingeniería ha tenido un proceso de cambio durante la historia de la humanidad. Desde los bocetos que fueron pioneros al planear antiguas construcciones o el funcionamiento de algún mecanismo, se ha intentado tener un registro de la creatividad e ingenio de los primeros diseñadores. Siendo que el hombre se ha preocupado por plasmar las ideas en algún sustrato, como lo es el papel, los planos arquitectónicos, estructurales, mecánicos o cualquiera que describiera algún proceso de construcción, logística o detallado han servido en el desarrollo de proyectos dentro de estas áreas. Debido a esto, los diseñadores que propusieron el uso de estas herramientas sabían que quizá un modelo tridimensional daría una explicación más detalla-

da de una construcción, pero la complejidad en cuanto al trazado de figuras tridimensionales y su comprensión para la construcción es muy alta, es por ello que se planteó usar una metodología estándar, la cual se caracteriza por una proyección en un plano. En la práctica profesional es común que se prefiera usar planos que muestren una proyección de una edificación, como por ejemplo "la planta" del edificio, la cual es una vista superior, o "los alzados" los cuales son cortes perpendiculares a la base de la edificación. Y esto ocurre debido a que es más sencillo trazar líneas en un plano que figuras en 3D, sin embargo, la simplificación en el diseño inicial provoca una demora de tiempo posterior al hacer el manejo de la información que ya está contenida en estos planos, debido a que los cambios que surgen durante la construcción tienen que ser trazados nuevamente en el mismo contexto de dibujo bidimensional.

En la primera imagen 2.1 se observa un plano arquitectónico de 1874, referente a la Iglesia St James en Littleton Inglaterra. Los planos arquitectónicos se caracterizan por mostrar los detalles finales de las construcciones, por ejemplo aquí se muestra el acabado final de la iglesia, y en la parte superior se muestra la distribución de espacios propuesta, la cual es un trabajo importante realizada por los arquitectos quienes hacen un gran empeño en aprovechar al máximo cada área disponible. Ahora, la imagen 2.2 representa un plano estructural de 1896 de un faro localizado en Moreton Bay. A diferencia de los planos arquitectónicos, los planos estructurales contienen información relevante a los detalles técnicos e incluso el procedimiento para la construcción, todo en base a cálculos previamente realizados que fundamentan la seguridad de la estructura. Estos siempre son representados en planos bidimensionales, debido a que el trazado 3D se vuelve muy complicado por la información a presentar. El estándar de trazado en planos se ha utilizado por siglos, y aunque algunas veces se trazan isométricos para representar el objeto en 3D, posteriormente se deben trazar planos bidimensionales para su construcción y traspaso de información entre los constructores. Esto era un proceso muy lento, ya que los trabajos realizados a mano tenían que ser trazados a lápiz pero cuando la revisión era aceptada se necesitaba utilizar tinta permanente para que los planos tuvieran un mejor aspecto. El problema es que la falta de práctica o incluso errores espontáneos utilizando estas técnicas causan que la tinta manche los planos, lo cual era común y desesperante porque se tenía que comenzar desde cero. Pensar que esto se realizaba para las grandes construcciones de inicios de 1900 es algo de increíble valor, ya que edificios como el *Empire State* o *Chrysler* necesitaban de salones repletos de personas que se dedicaban exclusivamente al trazo de estos planos.

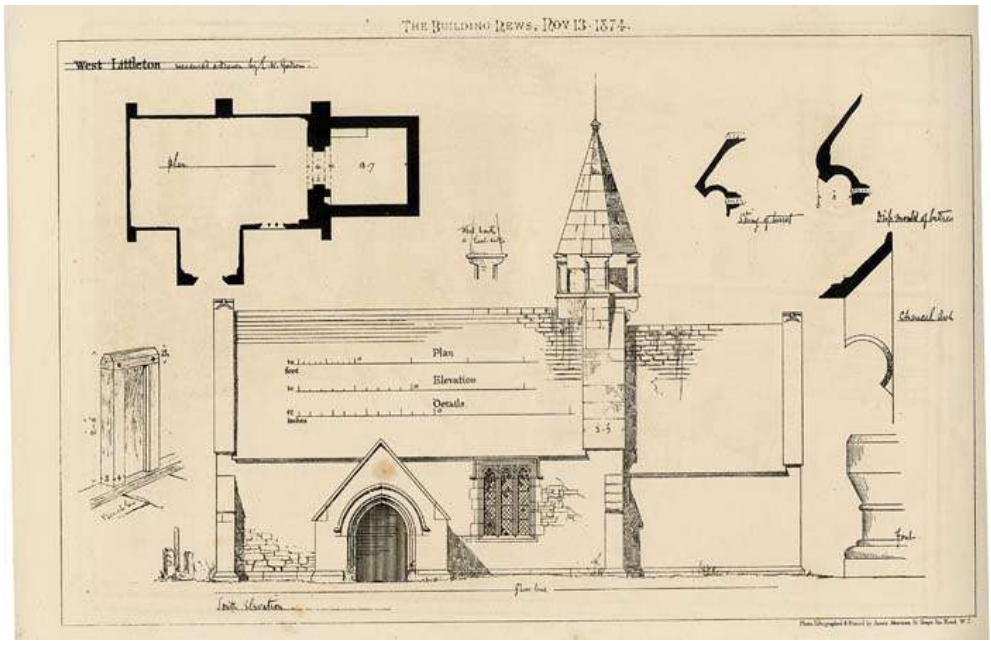


Figura 2.1: Contexto Histórico. Plano que muestra el diseño arquitectónico de la iglesia de St. James en West Littleton, Inglaterra. Realizado en el año 1874 (50).

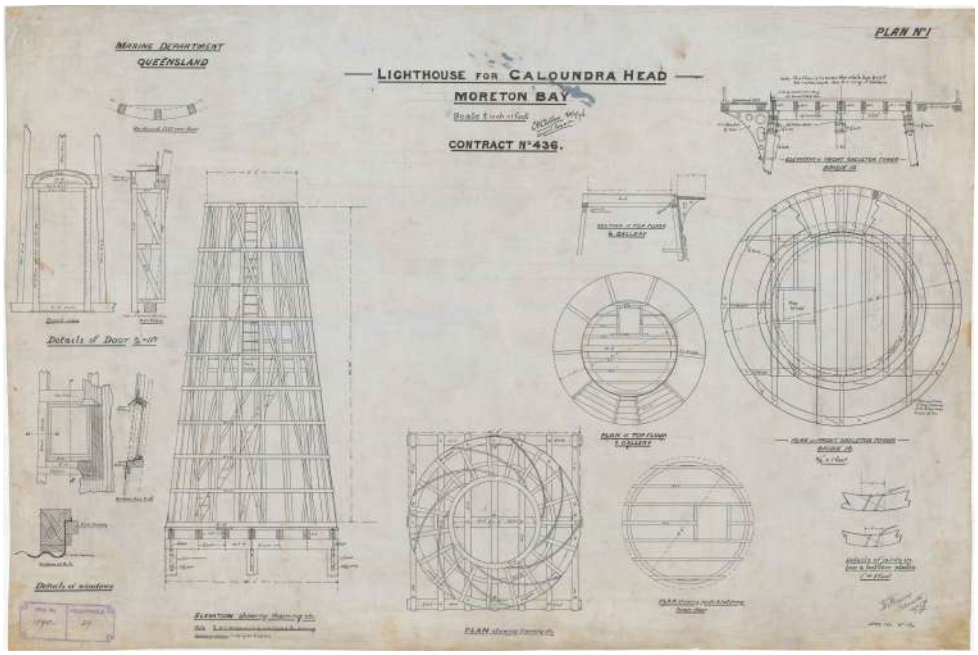


Figura 2.2: Contexto Histórico. Plano constructivo de un faro en Moreton Bay de Queensland, Australia realizado el 1896 (45).

Posteriormente se presentó la revolución de las computadoras y hasta la década de 1960's la interacción entre las computadoras era de forma numérica y de información escrita en papel. Científicos del Tecnológico de Massachusetts entre los que destaca Ivan Sutherland desarrollaron un sistema gráfico entre los humanos y la computadora, el cual era una forma de comunicación que permitía al usuario trazar figuras sobre una pantalla instalada en la computadora denominada $TX-2$ que fue diseñada en 1956, éste era un ordenador muy avanzado a su tiempo: tenía 320kb de memoria principal, un dispositivo de almacenamiento en cinta magnética de 8Mb, un monitor 1024x1024 de 7 pulgadas, un lápiz óptico y una caja de botones. Igual que la mayoría de ordenadores del momento, los programas eran escritos en macro ensambladores, gravados en unas cintas perforadas y alimentados en el lector principal del ordenador. La TX-2 ocupaba casi 93 metros cuadrados y, solo la memoria principal, aproximadamente 0.76 metros cuadrados. Estos aspectos conformaban la interfaz sobre la cual Ivan Sutherland basó su programa. El desarrollo del Sketchpad formaba parte de su tesis doctoral en el MIT, *"Sketchpad: A Man-Machine Graphical Communications System (1963)"*. Diseñó un sistema que permitía a los usuarios dibujar puntos, segmentos de líneas y arcos circulares, cruzar las líneas en un ángulo concreto y duplicar dibujos conservando una relación con el original (si este era modificado, los duplicados cambiaban automáticamente). El lápiz óptico era utilizado para dibujar directamente en el monitor del ordenador, e incorporaba técnicas gráficas de interfaz de usuario como la capacidad de borrar líneas y el zoom. El Sketchpad demostró que los ordenadores podían automatizar tareas repetitivas de diseño y borrador con una fiabilidad y precisión imposible de conseguir con métodos manuales. Una de las principales novedades del Sketchpad era la comunicación visual entre usuario y ordenador. Como el propio Sutherland dijo:

"En el pasado, hemos estado escribiendo cartas más que dialogando con nuestros ordenadores. El sistema del Sketchpad, por medio de la eliminación de las declaraciones escritas a favor del dibujo de líneas, abre las puertas a una nueva era de interacción persona-ordenador." - Ivan Sutherland

Pero no fue hasta la década de los 80's que las computadoras fueron más accesibles para las personas de la industria, en el área de la ingeniería automovilística se desarrolló el primer sistema comercial de dibujo asistido por computadora o CAD (por sus siglas en inglés). Lo cual trajo muchísimas ventajas para la realización de planos, ya que en lugar de muchas herramientas de trazo, solo se ocupaba una computadora. Se podía dibujar, modificar y borrar elementos con algunos comandos, por lo que rápidamente fue adoptado este sistema.

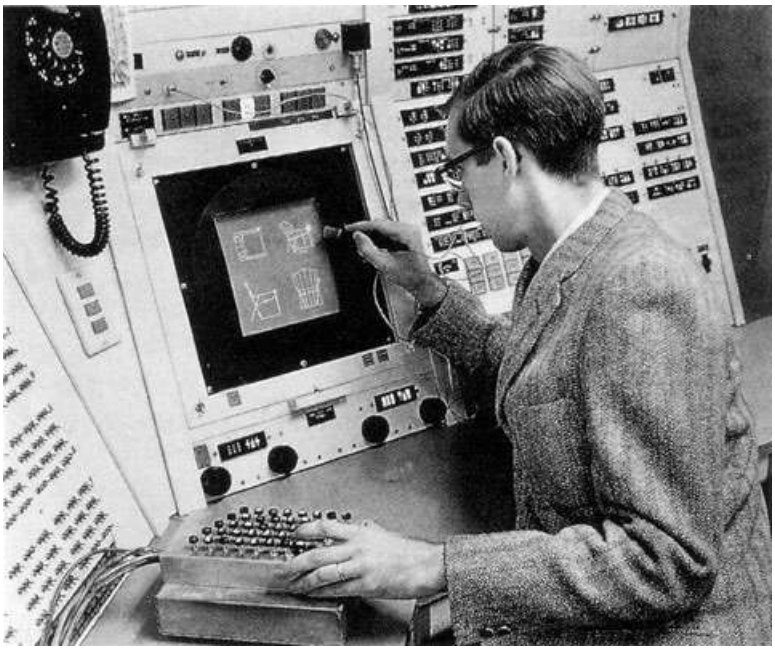


Figura 2.3: Contexto Histórico. *Sketchpad:* Sistema de comunicación gráfico entre humano y máquina (48).

Figura 2.4: Contexto Histórico. Primeros sistemas CAD aplicados en la industria mediante la accesibilidad a la computación personal (24).

Aún cuando la modelación de sólidos fue posible durante la misma época, los recursos computacionales no fueron suficientes para generar modelos tridimensionales eficientes que fueran sencillos de manejar por arquitectos y que a su vez afrontaran la necesidad del tiempo en los proyectos. Por ello se recurrió a continuar con el mismo estándar propuesto en el dibujo bidimensional, aunque esta innovación sí ayudo en el ahorro de tiempo al realizar el trazo en una computadora (CAD). No obstante, la ganancia se obtuvo en la misma etapa que antes, el diseño inicial es más rápido, pero la gestión de información posterior es casi de la misma magnitud que el trazado sobre papel, y esto es porque los trazos realizados en la computadora siguen teniendo propiedades geométricas sin reglas que permitan realizar modificaciones, sino que al momento de modificar un elemento, es necesario borrar y volver a trazar.

2.1.1. Sistemas de Modelación de Sólidos

Las áreas que comenzaron a utilizar el diseño asistido por computadora tridimensional (CAD) fueron las industrias de manufactura y producción, además de aeroespacial y mecánica en principios de 1970. Estas industrias vieron un aprovechamiento considerable en cuanto al uso de representación de geometría implementando algún análisis (estructural, térmico, acústico, etc.) que permiten una producción automática. Sin embargo, como se ha mencionado antes, el trazado de formas 3D dentro de un ambiente CAD tradicional de una parte mecánica es muy complicado y con tendencia a equivocarse, debido al uso de intersecciones de superficies para la unión de piezas. A mediados de la década de 1970, se hicieron investigaciones de acuerdo al modelado de sólidos tridimensionales, en donde destaca el trabajo de Braid (7), Requicha and Volecker (35), entre otros que fueron pioneros en esta área (citados por Sacks (41)).

El modelado de sólidos permitió que muchas metas que se proponía el CAD tridimensional fueran posibles, como por ejemplo: la representación de formas tridimensionales; la medición de cualquier tipo de forma tridimensional, incluyendo la obtención de volúmenes y áreas de superficies; el corte de secciones, la cual permite la integridad en las propiedades de dichas secciones; la generación automática de trazados; la verificación automática de interferencias espaciales entre geometrías; entre otros. Éstos desarrollos permitieron el control numérico de máquinas para la producción automatizada (41).

La modelación de sólidos permite hacer operaciones de edición como lo son la adición o sustracción de formas con otras, mientras se siguen preservando las propiedades de dichos volúmenes (4)(27) (citados por Sacks (41)). Los primeros sistemas CAD que incorporaron el modelado de sólidos se trajeron al mercado de la Arquitectura, Ingeniería y Construcción (AEC por sus siglas en

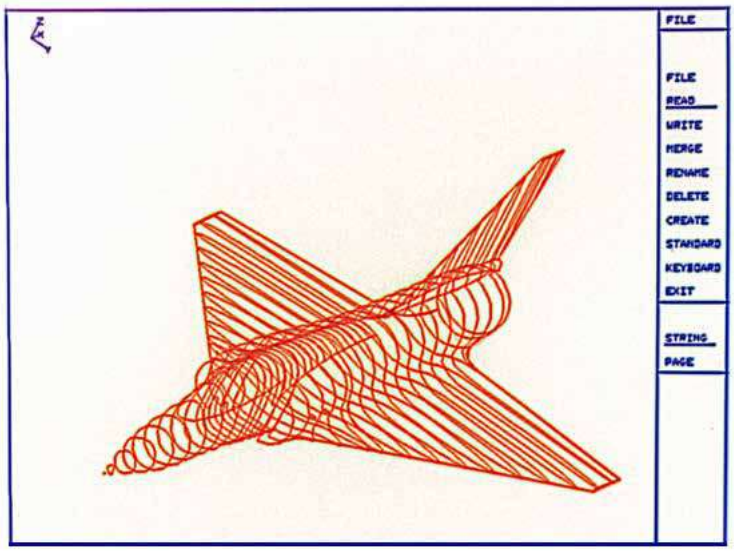


Figura 2.5: Modelación tridimensional. Primeros sistemas de modelado tridimensional utilizado en la industria aeroespacial. Software CATIA (49)

inglés) en la década de 1980, ejemplos de ellos son el denominado RUCAPS, Calma, TriCAD y PDMS, pero no tuvieron el impacto esperado. Esto se debió a que su uso era complicado, requería un enfoque de diseño distinto al que los diseñadores de esa época ya estaban acostumbrados a trabajar, no eran muy confiables y necesitaban de un poder de cómputo considerable para ese tiempo. Por lo tanto se encontraron con algunas desventajas como las mencionadas a continuación:

- No fue muy sencillo de usar al intentar diseñar modelos muy grandes, como por ejemplo un edificio.
- Las revisiones de los modelos sólidos mediante un CAD tradicional pueden ser tediosas, las operaciones geométricas requieren de múltiples pasos.
- Por tanto, el esfuerzo de hacer modelos tridimensionales se compara con el trazado de planos bidimensionales con sistemas más simples y baratos.
- Aunado a esto, se tenían siglos de utilización de un sistema estandarizado para la representación de información en construcción, el trazo bidimensional era ya una práctica común que imposibilitaba hacer un cambio drástico en la manera de planear.

Es por ello que la industria de la construcción y el diseño de proyectos determinó que era preferible utilizar los sistemas CAD bidimensionales. (41)

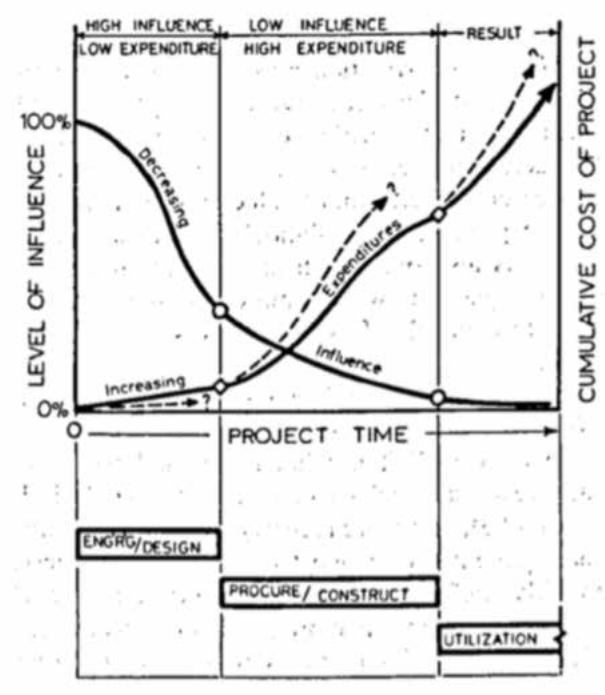


Figura 2.6: Curva de Paulson. Paulson indica los beneficios de la toma temprana de decisiones en un proyecto mediante la planeación de variables clave (9).

2.1.2. Modelación Mediante Parámetros

Paulson en 1976 propone que las decisiones importantes de diseño se realicen al principio del proyecto, ya que mediante una curva que trazó muestra que si el esfuerzo de diseño es mayor en la primer etapa, los cambios en etapas posteriores serán menos costosos (9). De manera contraria, si se comienza con un diseño muy básico, con menor esfuerzo inicial, el costo de modificación en etapas posteriores como la ejecución del proyecto o la fase operativa será por mucho más alto.

De acuerdo a la experiencia de los diseñadores, algunos proyectos pueden ser planteados de manera que con anterioridad se conozcan las variables clave y las fases que causan un retraso en la ejecución de los mismos, de este modo se puede contar con un prediseño más robusto que pueda amortiguar las modificaciones importantes originadas por los constantes cambios que se presentan durante la construcción física del modelo.

Similar a las ideas que presentó Paulson, Macleamy propone que las decisiones de diseño de un proyecto se deben concentrar en el inicio del mismo. Se debe recabar la suficiente experiencia para poder utilizar el mayor esfuerzo de planeación y diseño en la primer etapa, ya que en ese punto los cambios son menos costosos que al final de la ejecución del proyecto. A esto se le conoce como *"Front-Loading"* o por su traducción *"Carga Inicial"*, y los menciona Ma-

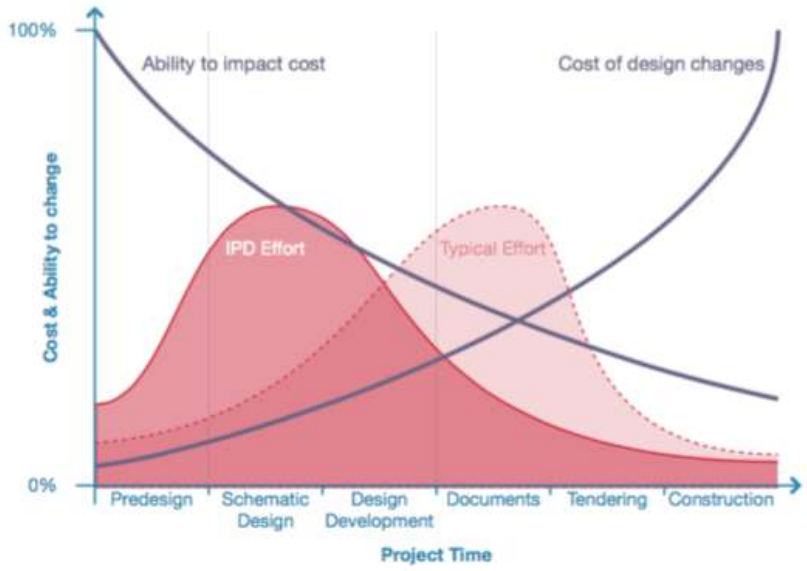


Figura 2.7: Curva de Macleamy. Macleamy, al igual que Paulson, piensa que la etapa principal de esfuerzo de diseño se debe centrar al principio del proyecto. Los diseñadores deben trabajar más cuando sus decisiones tengan mayor impacto y menor costo. (9)

cleamy junto con los siguientes dos conceptos: *"Entrega de Proyecto Integral"* o *IPD* (por sus siglas en inglés), que fusiona de manera estipulada las partes de proyectos para dar pie a los diseñadores sobre soluciones que pueden realizar en e principio del proyecto y que pueden ser viables desde el inicio; y el concepto que se está utilizando mucho últimamente en México, *"Modelado de Información de la Construcción"* o *BIM* (por sus siglas en inglés), la cual provee una base de datos central para mejorar la comunicación entre los miembros del equipo de trabajo, mientras que ayuda en algunos etapas como la simulación del proceso constructivo y posteriormente en la fase de documentación del proyecto.

Samuel Geisberg alienta a la exploración del diseño mediante parámetros diciendo, *"La introducción del modelado paramétrico fue motivado por el deseo de reducir el costo de los cambios. El objetivo es crear un sistema que pueda ser suficientemente flexible para alentar al ingeniero y que fácilmente considere una diversidad de diseños. El costo de hacer cambios de diseño debería ser tan cercano a cero como sea posible."* (citado por Davis (9))

El comentario de Geisberg y su pensamiento dan otro punto de vista respecto al esfuerzo requerido en un proyecto que Macleamy y Paulson representan en sus curvas. Él piensa que el objetivo no solo es mover el esfuerzo de diseño al principio del proyecto ingenieril sino hacer lo posible para que "el costo de hacer cambios de diseño sea tan cercano a cero como sea posible". En teoría, un modelo basado en parámetros ayuda a reducir los pasos para realizar cambios, ya que cuenta con funciones explícitas que sirven como herramienta para el modelado. (9)

Tomando como referencia las ideas anteriores, podemos comentar que si los cambios pueden ser anticipados, el modelo puede se construido únicamente con los parámetros necesarios para alojar dichos cambios. El problema existe cuando un cambio no es anticipado, entonces el diseñador debe adaptar la modificación realizando algún cambio en las funciones o variables clave del modelo. Es entonces que la tarea de construir un modelo mediante parámetros así como modificar funciones establecidas en el inicio puede ser muy desafiante y más aún cuando el modelo es complejo, como lo puede ser una edificación integral (46). Si el problema es definir las variables clave con anticipación, la solución más obvia es proveer mucha flexibilidad desde el inicio. Es decir, hacer que cada parte del modelo sea lo más flexible, con muchos parámetros que puedan ser utilizados por el diseñador. Pero el uso de parámetros vienen con un costo, se requiere mucho trabajo planear y anticipar todos los "caprichos" de un diseñador, además el modelo se vuelve tan complejo que su utilización no es práctica.

"El desafío de construir un modelo paramétrico es desenredar las interdependencias creadas por los diferentes requisitos y encontrar un conjunto de reglas que sean sencillas y que permanezcan lo suficientemente flexibles para adaptarse a cualquier caso. En otras palabras: determinar con precisión la perspectiva a un nivel exacto de abstracción donde ningún punto importante sea perdido y no se desvíe el objetivo por detalles innecesarios." (43)

Matemáticos y científicos han usado el término en relación a varias representaciones geométricas. Sin embargo, los arquitectos han adoptado el modelado paramétrico como una expresión de diseño, por tanto la definición de *paramétrico* se ha vuelto turbia. Ahora cuando los arquitectos usan este término, ellos pueden referirse a todo el diseño, o solo a los diseños que cambian, o a las herramientas, o al estilo de diseño. Este desacuerdo existe cuando se define paramétrico por autores prominentes en muchos de sus trabajos. Para los arquitectos, los modelos paramétricos mejoran la habilidad del diseñador para hacer cambios, de este modo mejoran su capacidad de diseño. En teoría un diseñador puede modificar los parámetros de un modelo y ver como cambia el diseño casi instantáneamente. Así, un modelo paramétrico ideal abarca todas las variaciones que el diseñador necesite explorar con el más conciso conjunto de parámetros posible. Por otra parte, un modelo paramétrico puede ser tan único cómo sea planteado, y no por el hecho del uso de parámetros (ya que todo diseño por simple definición, tiene parámetros asociados), no porque cambia (otras representaciones del diseño tienen la habilidad de cambio), y no porque se considera como una herramienta o estilo de arquitectura. ***Un modelo paramétrico es único no por lo que hace, sino por cómo fue creado. Un modelo paramétrico es creado por un diseñador explícitamente mediante reglas que indicarán como los resultados se derivan de un conjunto de parámetros.***

Figura 2.8: Diseño Paramétrico en Arquitectura. El Centre Pompidou-Metz es un museo de arte moderno y contemporáneo, se observa la arquitectura de tipo orgánico y que por sus características estéticas se le clasifica conceptualmente dentro del diseño paramétrico (56).

2.2. Diseño Paramétrico Computacional

El diseño paramétrico pretende obtener mayor eficiencia en cuanto al manejo de la información y ofrece una opción moderna a los procesos tradicionales. Según la *Enciclopedia Concisa de las Matemáticas* (54), **paramétrico** significa "Conjunto de ecuaciones que expresan un conjunto de cantidades a partir de funciones explícitas de un número de variables independientes, conocidas como "parámetros"". Dicha definición matemática puede ser llevada al contexto que estamos manejando, por lo que Davis menciona que "Un modelo paramétrico es un conjunto de ecuaciones que expresan un modelo geométrico a partir de funciones explícitas de un número de parámetros" (9).

Shah describe al diseño paramétrico como al modelo que usa restricciones geométricas y formulaciones matemáticas de interdependencias entre ellas mismas (22), su base es la generación de formas geométricas a partir de una familia de parámetros iniciales y programación de las relaciones formales que guardan entre ellas. El diseño paramétrico utiliza variables y algoritmos para generar un árbol de relaciones matemáticas y geométricas que permitan no sólo llegar a un diseño, sino generar todo el rango de posibles soluciones que la variabilidad de los parámetros iniciales nos permitan (15).

Debido a la complejidad de las herramientas utilizadas por los diseñadores para modelar geometrías en tres dimensiones y al costo computacional necesario, se buscaron alternativas que permitieran desarrollar modelos más rápidamente y con un esfuerzo considerablemente menor. Entonces, utilizando la representación de formulaciones algebraicas para generar sólidos primitivos (p. ej. cilindros o cubos), los cuales pueden ser combinados utilizando operaciones como la unión, intersección y sustracción mediante funciones, se obtiene una representación compacta y fácil de editar (34) (citado por Sacks (41)). Estas expresiones se conocieron como geometrías sólidas constructivas (CSG por sus siglas en inglés), y diverge del modelado de sólidos clásico de superficies de frontera, denominado representación de frontera (B-rep por sus siglas en inglés). Posteriormente, se añadió la capacidad de hacer referencia explícita a los parámetros que definían las operaciones, esta técnica fue pionera en el desarrollo de lo que hoy conocemos como *sistemas de modelación paramétrica de sólidos* (23) (citado por Sacks (41)). Este concepto de modelado paramétrico no era del todo nuevo, algunas de sus capacidades fueron incorporadas en los primeros sistemas CAD, como lo es el desarrollo conocido como *Sketchpad* (48).

La parametrización se expandió rápidamente debido a la facilidad del uso para la modelación de sólidos. Las reglas impuestas en el diseño paramétrico actuaron como restricciones y éstas fueron añadidas a los primeros modeladores paramétricos de sólidos, de este modo se guió la regeneración del modelo en vista de diferentes condiciones de diseño. Entonces una cuestión que debe ser remarcada es que los sistemas paramétricos no solo generan formas representados en imágenes, sino que las restricciones y reglas utilizadas para generarlas también son mantenidas en todo momento como parte integral del modelo. Esto permite una expresión de diseño con comportamiento dinámico intencional. Se pudiera decir en este contexto que el *comportamiento* se refiere a la respuesta de *"un sistema automático de acciones que mantienen una consistencia topológica y geométrica de las relaciones dentro del objeto principal y entre los objetos que interactúan"*. El modelado paramétrico utiliza funciones para construir objetos desde cero y refinar ensambles complejos con cierta facilidad. Algunas de las operaciones típicas que son aplicables a figuras dentro de un contexto paramétrico son "Intersección de figura A con figura B", y al realizarse se consideran que se encuentran topológicamente relacionadas. Otro ejemplo de operación en un mismo objeto, es decir, una operación unaria, sería: "Chaflán en una esquina", y esta se activaría realizando un corte en un cuerpo dependiendo los parámetros requeridos. Hay que tener una especial atención en las reglas impuestas, ya que la evaluación de un modelo puede tener complicaciones de acuerdo a las restricciones realizadas. Un modelo que tenga conflicto entre dos o más restricciones se conoce como *Modelo Sobredefinido*, y no se generará debido a que las restricciones dadas impiden su creación. Por otra parte, un resultado que no es único es conocido como *Modelo Subdefinido*, por lo que se necesitará que alguna restricción sea más estricta.

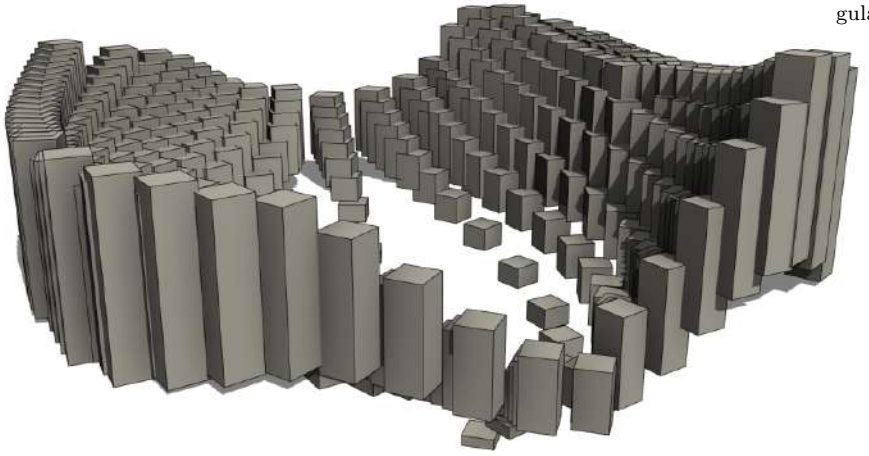


Figura 2.9: Håvard Vasshaug - Modelado Paramétrico. Se muestra un objeto geométrico generado por una distribución de prismas rectangulares (19).

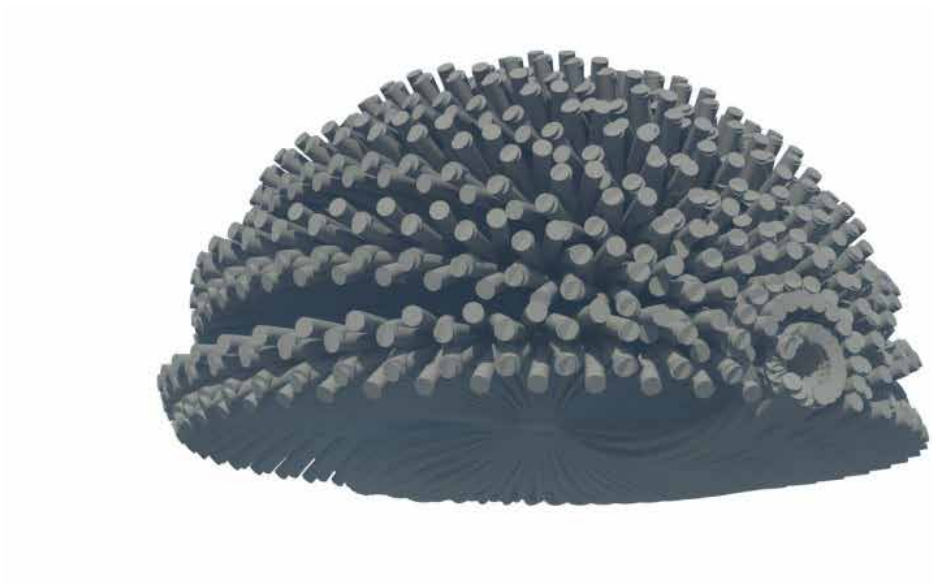


Figura 2.10: Håvard Vasshaug - Modelado Paramétrico. Objeto cuyas componentes representan un reto de parametrización debido a su complejidad (19).

El modelado paramétrico hace una contribución significativa al diseño, que al igual que en el modelado de sólidos, permite que se generen representaciones computacionales de objetos físicos no solo como una imagen, sino con la manera de definir las relaciones semánticas entre las representaciones de los objetos, permitiéndoles ser creadas y editadas fácilmente. Una de las personas que me ha inspirado por su trabajo ya que a mi parecer hace una contribución al diseño con metodologías computacionales es Håvard Vasshaug. Él es un diseñador tecnológico que trabaja en la agencia de arquitectura y diseño "Snøhetta", además que tiene una consultoría propia. Su formación como Ingeniero Estructural y su pasión por la cual el mundo digital 3D está en su mente, le permitió trabajar en modeladores tridimensionales comerciales como lo son Revit, Rhino, Dynamo y Grasshopper. Håvard cuenta con una página en línea con temas acerca de modelado paramétrico y ejemplos que ha realizado para mostrar los alcances de esta nueva metodología. Su página, de tipo "blog" con dirección `https://vasshaug.net/`, cuenta con una sección llamada "galería", en la cual muestra trabajos muy interesantes sobre modelado tridimensional. Las imágenes 2.9 y 2.10 fueron extraídas para mostrar las capacidades y alcances que los diseños pueden tener. Puede pensarse que al utilizar modeladores paramétricos basados en generar geometrías mediante funciones, el diseño se encuentra limitado, pero como se puede apreciar, es la creatividad del diseñador quien utiliza estas herramientas da la flexibilidad necesaria para obtener resultados impresionantes.

El software de modelado paramétrico dedicado a un dominio específico de diseño permite formular relaciones y restricciones especializadas para expresar la lógica dentro del contexto que se está manejando. Es útil generar una biblioteca de formulaciones paramétricas que permitan diferenciar las diferentes aplicaciones. El uso de la tecnología y relaciones previamente definidas habilitan la posibilidad de definir modelos complejos mediante ensambles. Un ejemplo de esto son los modelos de edificaciones, ya que por sus características constructivas y de planeación el número de parámetros puede ser infinito. En este caso del diseño de edificios, la geometría puede ser derivada de relaciones espaciales entre los elementos que la conforman. De esta manera, los cambios que se hagan a cualquiera de los parámetros que definen la edificación, se propaga a todos los elementos automáticamente, denotando un cambio completo de la geometría inicial (41). Håvard muestra un modelo de edificación en la imagen 2.11, la cual puede tener funciones específicas para su diseño, como por ejemplo la utilización de refuerzo y presfuerzo en losas que el mismo Håvard modela y crea aplicaciones, un ejemplo se muestra en la figura 2.12. La utilización de software de modelación tridimensional en conjunto con un sistema de modelación paramétrico permite al diseñador crear y modificar de forma más eficiente las propuestas en un proyecto. Este es el caso de la edificación de tipo orgánico que el museo del Centre Pompidou-Metz tiene. Y ya que en la arquitectura moderna se tienen corrientes de diseño a las cuales se les ha dado

Figura 2.11: Håvard Vasshaug - Modelado de Edificaciones. Edificación generada mediante un software de modelado tridimensional (19).

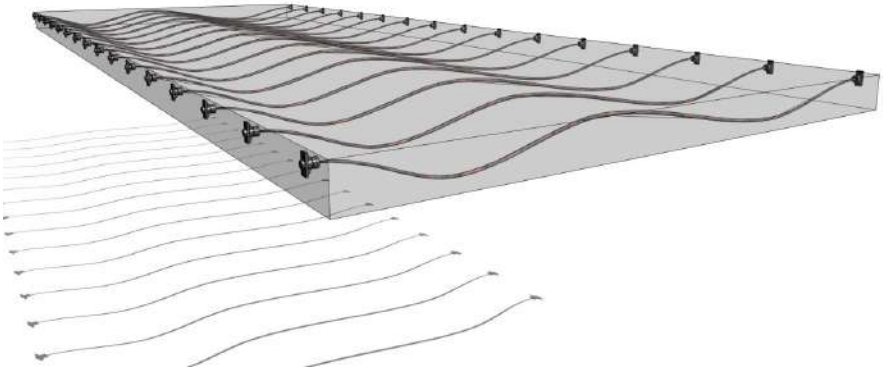


Figura 2.12: Håvard Vasshaug - Modelado de Edificaciones. Representación tridimensional del refuerzo en una losa (19).

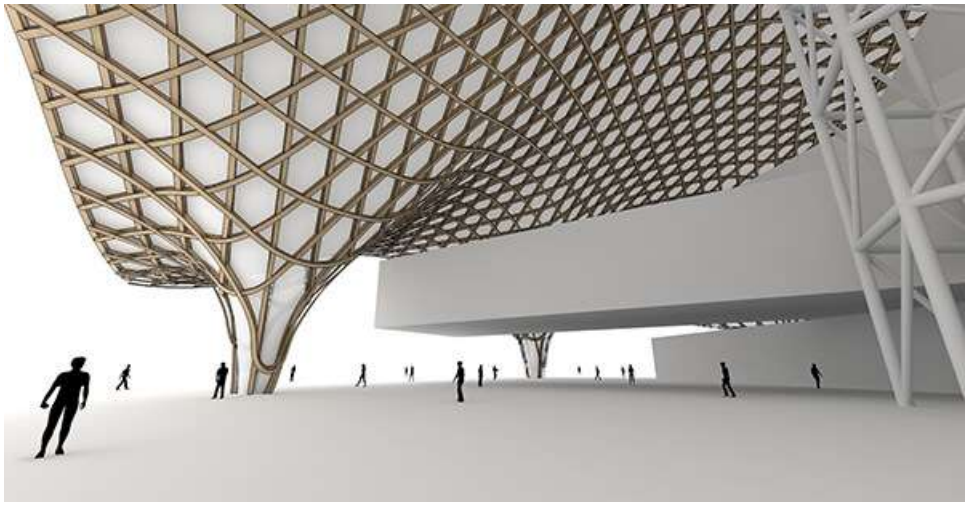


Figura 2.13: Diseño Paramétrico Computacional. Se muestra el modelo del Centre Pompidou-Metz utilizado como materia prima para un curso de herramientas tecnológicas de parametrización, el modelo es un render con propiedades paramétricas utilizadas para obtener un mejor desempeño de sus características estéticas y de diseño. La construcción real se muestra en la imagen 2.8 (56).

el nombre de "Arquitectura Paramétrica" por la características que cuenta, la mejor opción para realizar propuestas dentro de este estilo es utilizar las herramientas tecnológicas para definir las relaciones y reglas entre los elementos que interactúan. El modelo computacional realizado para la construcción de la figura 2.8 se muestra en la imagen 2.13.

2.2.1. Programación en el Diseño Paramétrico

Aunque hay claras diferencias entre hacer arquitectura y programar un software, la logística de tareas en ambos casos debe ser bien analizada para poder hacer más eficiente un trabajo. Y de acuerdo a lo mencionado antes, tenemos que para un arquitecto, hacer un diseño mediante las reglas de la parametrización es más similar a programar que a diseñar con un procedimiento convencional. Durante la década pasada se han visto muchos desarrollos en los modeladores paramétricos, pero los arquitectos se han dado cuenta que existe cierta complejidad en la creación de un modelo paramétrico simple. Pudiera decirse que se está tomando como acto de valor tomar esta metodología respecto a una tradicional, porque construir un modelo desde cero puede ser costoso y si a mediano plazo del proyecto éste cuenta con deficiencias iniciales, la preocupación comienza y la decisión de tomar otro camino (uno tradicional) puede llegar a ser más viable . Saber si es buena decisión un modelo paramétrico es una preocupación similar a la que se enfrentaron diversos científicos en una reunión que organizó la OTAN en 1968 sobre ingeniería de software. En esos años se observaba la tendencia del acercamiento de la computación a la sociedad y las actividades humanas. Se tenían observaciones que el costo de hacer un software se estaba comparando con el costo de construir el hardware de una computadora. Esto era alarmante desde el punto de vista que muchos códigos de programas que tenían un futuro remunerante, fallaban de manera sorpresiva y el tiempo invertido se convertían en gastos perdidos. Uno de los factores pudo

ser que cada vez se elaboraban programas más complejos, los cuales llegaban a un punto tan especializado que perdían la flexibilidad de su uso, y esto aunado a la creciente aparición de los denominados *"bugs"* o fallas en el código, ayudaron a que se buscaran procedimientos de medición de calidad de software (9).

Un científico destacado sobre ingeniería de software de la época, fue Barry Boehm, quien no asistió a la conferencia de la OTAN, pero claramente fue influenciado por dichos temas. Boehm elaboró un artículo en donde comprobó sobre el costo de la producción de software y demostró mediante gráficas cuantitativas que éste se estaba volviendo más costoso que la propia manufactura de hardware. En su artículo muestra una curva, que hizo famosa su publicación, denominada *La Curva de Boehm* (figuras 2.14 y 2.15), en donde se observa que mientras un programa se vuelve más desarrollado, también se vuelve más difícil de modificar. Este mismo comportamiento se compara con la *Curva de Paulson* vista en la figura 2.6 de la sección 2.1.2, la cual esta basada en observaciones de proyectos arquitectónicos, y se puede observar que poseen la misma tendencia. Sin embargo la mayor diferencia entre estas dos curvas es que Boehm utilizó mediciones reales, mientras que Paulson simplemente utilizó un planteamiento propio basado empíricamente en su experiencia. Los datos de Boehm pronostican que haciendo un cambio en una etapa tardía del proyecto de software, ésta pudiera costar cien veces más hacer el mismo cambio cuando se inició el proyecto. Además de las curvas de Paulson y Boehm, ya existen muchos otros estudios basados en estos que demuestran el comportamiento del desarrollo de cualquier tipo de proyecto, sin embargo este trabajo solamente tomará estos como referencia para el caso de estudio seleccionado.

Para una computadora, un modelo paramétrico es un conjunto de instrucciones. La computadora toma las entradas, aplica la secuencia de funciones mediante el uso de algoritmos y de este modo se genera como resultado el modelo paramétrico. La programación que se requiere en un campo como lo es el diseño de edificaciones, tanto arquitectónico como estructural, es preferida dentro de ambientes donde se pueda resolver el problema de manera integral junto con el desarrollo del proyecto. De entre los paradigmas de programación, tenemos que es común que en esta área se empleen en mayor medida lenguajes imperativos que están incorporados a plataformas CAD, y aunque han sido bien estudiados, al final son complicados de utilizar y por tanto se consideran limitados, dando como resultado que el ingeniero promedio desee seguir utilizando la metodología tradicional. Esto significa que se tiene un rango reducido de soluciones disponibles para expresar ideas dentro de un proyecto en construcción. *Esto presenta una oportunidad de expandir la práctica del modelado paramétrico utilizando nuevos paradigmas de programación que ayuden a incorporar a los ingenieros en el manejo eficiente del software.*

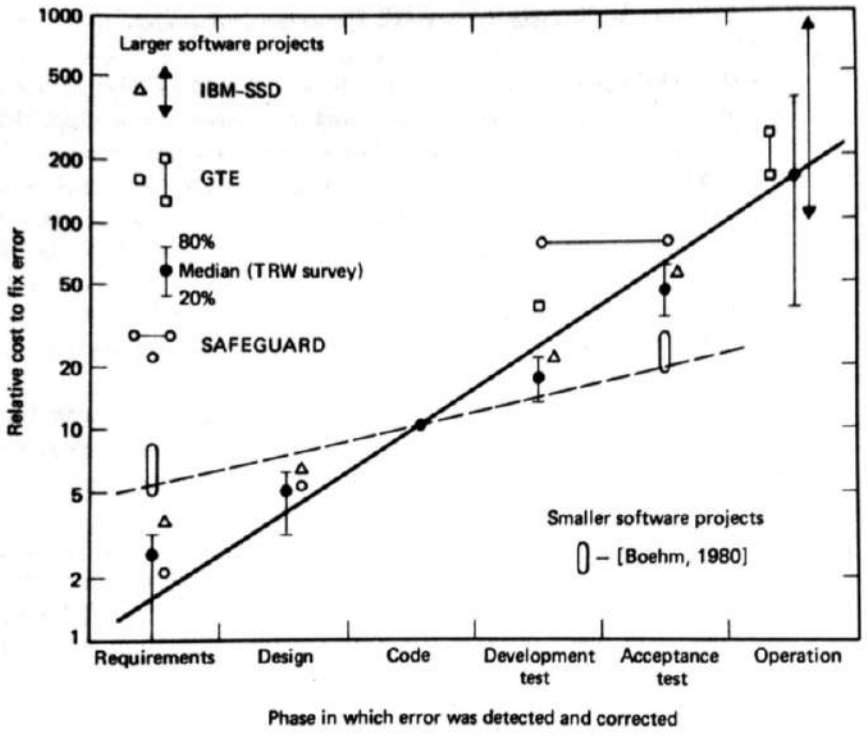


Figura 2.14: Curva de Boehm. Gráfico que muestra con datos reales, el comportamiento de un proyecto de desarrollo de software desde su diseño hasta su fase operativa. La curva está trazada con una escala logarítmica. ((6) citado por Davis (9))

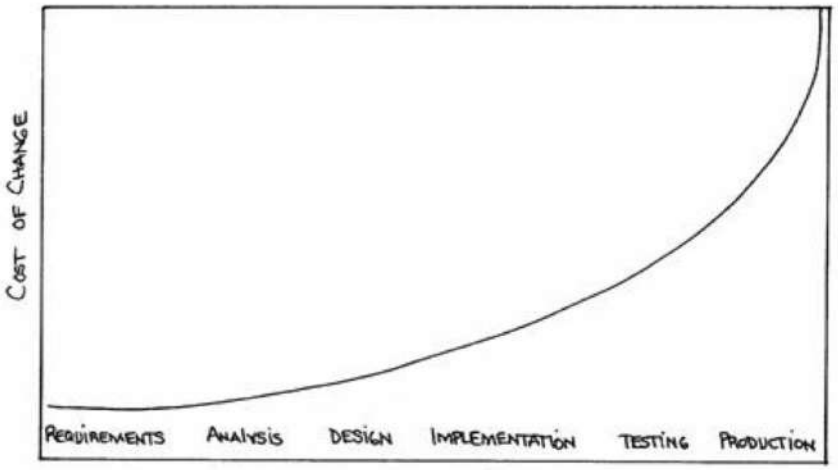


Figura 2.15: Curva de Boehm. Gráfico con escala lineal, la cual muestra un comportamiento similar a la curva de Paulson de la figura 2.6. ((6) citado por Davis (9))

David Rutten, desarrollador de **Grasshopper** (modelador paramétrico) menciona: *"cualquier persona involucrada en algún trabajo que escriba instrucciones que posteriormente serán ejecutadas por una computadora, máquina o incluso una entidad biológica, puede decirse que se encuentra programando "* (37) (citado por Davis (9))

Pienso que la frase de David Rutten es muy acertada y me pareció excelente motivación para que áreas en las que no se obtiene un *perfil de programador* pero que realmente es necesario automatizar tareas, hacer eficiente el trabajo y optimizar las propuestas, se acerquen a la programación y "pierdan el miedo" a utilizar nuevas tecnologías que no solo hacen que nuestra labor sea más rápida, sino que permiten resolver nuevos problemas y a enfrentar nuevos horizontes. En los proyectos de construcción por lo general se necesita resolver el problema rápido, por ello difícilmente los arquitectos e ingenieros se dan el tiempo para dominar un lenguaje de programación para hacer eficiente su trabajo, esto también se puede representar por las curvas de Paulson y Boehm, ya que si en la práctica se invirtiera tiempo inicial para conocer nuevas metodologías, el trabajo posterior no costaría tanto esfuerzo. Sin embargo, dado la gran mayoría de personas que aún no se ha podido acercar a realizar programación textual, surgió una alternativa dentro del conjunto de lenguajes declarativos conocido como "Dataflow" o flujo de datos.

Programación Mediante Flujos de Datos

John Sharp en "Data Flow Computing" (1992) define a los programas que usan el concepto de dataflow como "un lenguaje en el cual la ordenación de operaciones no está especificada por el programador, pero las operaciones se ejecutan mediante las interdependencias de información". En otras palabras, un lenguaje dataflow describe las conexiones entre operaciones computacionales, lo cual es diferente al enfoque imperativo de enlistar operaciones en el orden que deberán ocurrir. Al momento que se está ejecutando un programa basado en dataflow, la computadora infiere el orden de instrucciones que deberá seguir para hacer un óptimo manejo de los recursos, lo cual se define mediante las conexiones de datos realizadas por el usuario. Un ejemplo esencial de la programación dataflow, son las hojas de cálculo, ya que el usuario define la conexión entre celdas y como se procesa la información, pero deja a la máquina decidir el orden de actualización de celdas. El mismo principio es aplicado a ciertos software de modelado paramétrico, ya que los usuarios definen una red de operaciones geométricas, mientras la computadora maneja la secuencia y ejecución de operaciones. En este paradigma no es necesario hacer una programación textual como se acostumbra a programar en ambientes de desarrollo de software, es por ello que se le denomina *"lenguaje de programación visual"*. El uso de este paradigma es muy similar entre los lenguajes del mismo tipo, es decir, los elementos que contiene siempre son los mismos, se tiene que ha-

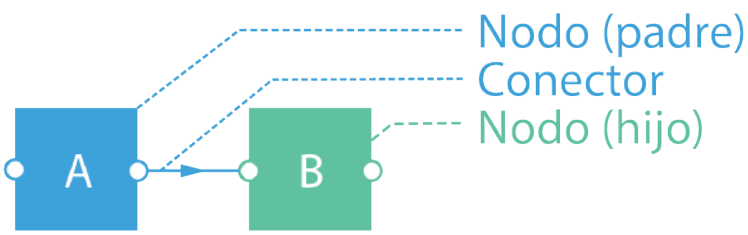


Figura 2.16: Data Flow Computing. Estructura general de un diagrama acíclico dirigido (DAG). Figura adaptada de (9).

cer relaciones de información mediante un gráfico denominado *"gráfico acíclico dirigido (DAG por sus siglas en inglés)"*(9). Este paradigma de programación permite al usuario una mejor visualización en cuanto a lo que está programando, debido a que los diagramas de flujo permiten observar la etapa que se está ejecutando, además al agregar una función nueva, se puede ver el cambio en el modelo resultante.

La estructura general de un DAG consta de módulos representados por cajas, denominados nodos, que hacen alguna función, en las que destacan las funciones geométricas en los ambientes de programación visual para arquitectos o diseñadores, los cuales tienen una entrada (input) en el caso más simple o más, dependiendo el tipo de función a utilizar, de la misma manera constan de una o más salidas (outputs). Para que una función sea activada es necesario alimentarla de información, esto se hace conectando un nodo en el input de dicha función, de esta manera se puede crear una red de conexiones para generar un modelo deseado. Al nodo que alimenta se le conoce como "nodo padre", en cuanto al nodo que recibe información para ejecutarse se le conoce como "nodo hijo". Se podrá pensar que utilizando estas funciones en lugar de usar un ambiente CAD tradicional pudiera ser más costoso en cuanto al esfuerzo invertido. Y quizá tenga sentido, ya que al usar un software con interacción tipo CAD tradicional donde solamente hay que seleccionar la función y utilizando el "mouse" del ratón se traza una línea, se vuelve un procedimiento más simple y sencillo que tener que hacer conexiones de nodos. Pero la idea no es sustituir una metodología de CAD mediante el uso de periféricos manejados por el usuario, mas bien es unir diferentes herramientas para complementarlas y realizar un trabajo de manera que se pueda tener un control completo de la problemática. Además, hay que pensar como lo hacían Paulson y Boehm, si consideramos una planeación con más variables clave iniciales, en la etapa de construcción y modificación, el esfuerzo requerido será menor.

Esto lo podemos ejemplificar con la figura 2.17 que tiene el crédito nuevamente de Håvard Vasshaug, en la que podemos observar varias cuestiones. Primero, el recuadro central tiene la red *DAG* utilizada. En segundo lugar podemos observar cualquiera de los otros recuadros, y veremos un panel generado en un ambiente tridimensional. Este panel puede ser cualquier componente de

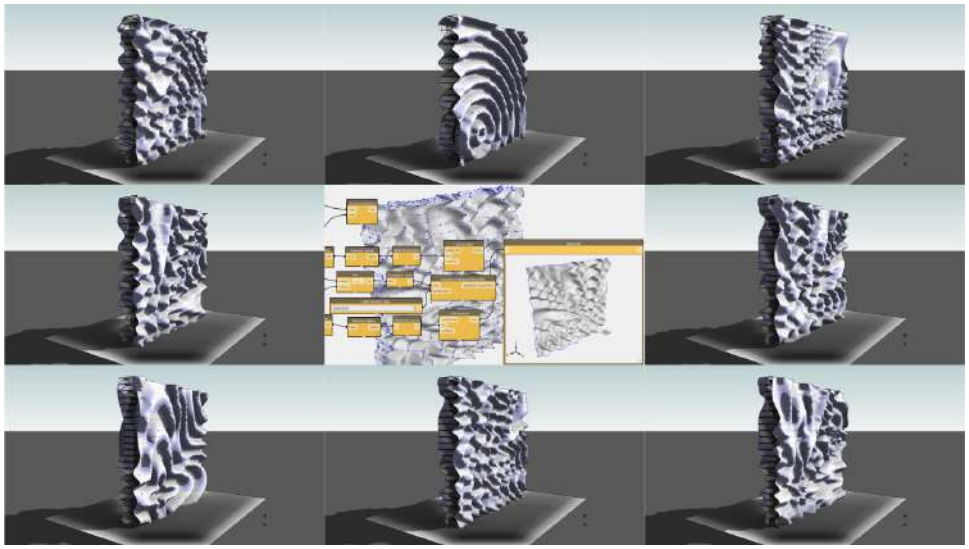


Figura 2.17: Múltiples diseños. Panel generado por módulos o nodos que se conectan mediante una red tipo DAG, la figura fue realizada como ejemplo de funciones trigonométricas en modelos paramétricos por Håvard Vasshaug (19).

diseño posible, quizá una fachada o simplemente un monumento. La idea es que utilizando la computación por flujos de datos o *"Data Flow Computing"*, es posible generar un panel y cambiando algún valor numérico que sirve como parámetro para las funciones en los módulos o nodos, automáticamente se crea un nuevo modelo. En este ejemplo las funciones utilizadas son trigonométricas por ello podemos ver los paneles con *ondas*. Este ya es un ahorro de tiempo considerable, ya que de manera contraria al planteamiento utilizado por Håvard, pensemos en cómo sería el procedimiento si quisiéramos generar un modelo como éste en un ambiente CAD tradicional. Seguramente generar el *panel número 1* (tómese cualquier recuadro como referencia) sería un poco tedioso, la cuestión es que si ahora quisiéramos generar el *panel número 2* (tómese un recuadro distinto como referencia), se tendría que borrar prácticamente todas las líneas y superficies que integran al *panel 1*, ahora si sería muy tedioso volver a empezar. Y esto es gracias a la disposición de información que se encuentra interconectada y la utilización del sistema de programación visual que dispone el modelador.

Sin embargo, una consideración crítica en cuanto a la programación visual mediante *dataflow* es el hecho que el flujo de datos tiene una dirección que se especifica por el programador. Esta característica la podemos observar en el DAG de la figura 2.18, donde se utilizan dos nodos "punto" los cuales dan como resultado un conjunto coordenadas en un espacio geométrico, éstos proveen de información a otro nodo "línea" que necesita de un punto inicial y uno final para poder ejecutarse, al hacer la conexión correcta podemos observar un elemento geométrico que ha sido generado mediante dicha relación. La dirección dada por el programador tiene implicaciones importantes, ya que además de especificar operaciones, también existe una jerarquía dadas las conexiones. Co-

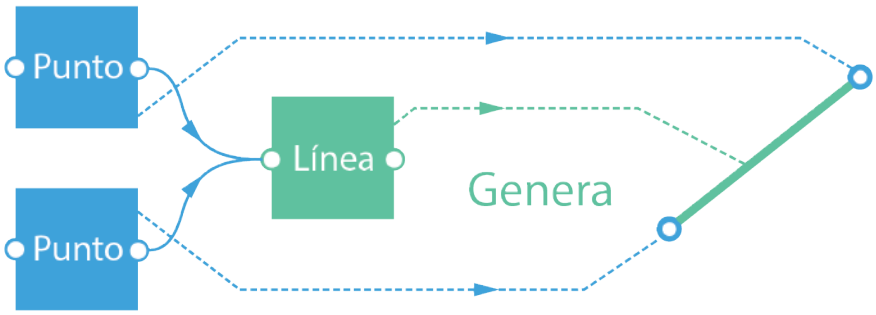


Figura 2.18: Data Flow Computing. Obtención de una geometría simple (línea) mediante un DAG. Figura adaptada de (9).

mo consecuencia, si el programador decide poner de reversa el flujo de datos, convirtiendo los hijos en padres, hay riesgo de quebrantar la estructura del programa (9). La verdad es que al hacer una comparación entre la facilidad de utilización del lenguaje de programación visual con respecto a uno textual, se encuentra que dichas jerarquías mencionadas en la programación visual son prácticamente cuestión de lógica simple. Si lo que se desea es generar una línea, se debe pensar que la línea necesita como *input* dos puntos, caso contrario sería querer generar una línea conectando su output a los parámetros de entrada de los módulos punto, esto no tendría sentido. Solo en algunos casos esta lógica se puede complicar y no es tan sencillo observar la jerarquía de conexión, pero siempre será recomendable revisar la documentación de cada función a utilizar para asegurar su correcto funcionamiento.

Sterling y Shapiro (1994) definen que la parte principal de "un programa lógico es un conjunto de axiomas o reglas que definen relaciones entre objetos". Incluso cuando desarrolladores creen que la programación lógica se convertirá en la herramienta del arquitecto del futuro, aún falta camino para que suceda. De manera puntual, Davis menciona que en su investigación no encontró proyectos reales complejos que hayan empleado esta metodología (9). En las siguientes secciones se presentarán algunos conceptos de programación que sirvieron como base conceptual para pensar en una herramienta que fusione la programación textual, la programación con flujos de datos y la programación visual.

Programación Estructurada

Lo más recomendable a la hora de escribir un código que tenga una aplicación definida es utilizar *módulos* que integren al programa en su totalidad. Un módulo es definido por Wong y Sharp ((44) citado por Davis (9)) como "una secuencia de instrucciones dadas por un punto de entrada y uno de salida" que realiza "un tarea relacionada con algún problema". Los módulos tienen cinco principales beneficios según Bertrand Meyer ((31) citado por Davis (9)):

- **Descomposición** Un problema complicado puede ser descompuesto en una serie de sub-problemas más simples, cada uno contenido con dentro de un módulo. Descomponer problemas de esta manera hace más fácil el enfoque y puede ser más sencillo para equipos al trabajar juntos ya que cada miembro puede trabajar en sub-problemas separados.

- **Composición** Si los módulos son adecuadamente autónomos, pueden ser re-combinados para formar nuevos programas (una composición). Esto habilita la funcionalidad dentro de cada módulo (o como se dirige un sub-problema) para ser compartido y rehusado mas allá de su contexto original.

- **Entendimiento** Si un módulo es plenamente independiente, un programador puede ser capaz de entenderlo sin necesidad de descifrar el programa completo. A la inversa, un programador debería ser capaz de entender todo el programa sin tener que leer los detalles de implementación de cada módulo. Dijkstra ((13) citado por Davis (9)) se preocupó ya que esto podría dirigir a reducir transparencia de los códigos pero muchos han argumentado que la abstracción ayuda al entendimiento. Por ejemplo, McCabe ((29) citado por Davis (9)) ha planteado que hacer módulos mejora el entendimiento ya que reduce la complejidad ciclomática, haciendo "un camino en el cual un programa puede ser controlado". Meyer ((31) citado por Davis (9)) apunta que hacer módulos ayuda a los programadores a comprender mejor el código a través de los nombres dados a las entradas, salidas y al módulo mismo.

- **Continuidad** Un programa tiene continuidad cuando los cambios pueden hacerse sin provocar cascadas de otros cambios. En un programa sin continuidad, cambiar un modulo sin afectar a los módulos dependientes, produce una reacción en cadena ya que todos los módulos dependientes son cambiados para acomodar el cambio original y así sucesivamente. Continuidad tiene mucho que ver como una estructura de un programa es descompuesta. David Parnas ((32) citado por Davis (9)) sugiere que los proyectos deberían fallar alrededor de "decisiones difíciles de diseño o decisiones que cambian fácilmente" por ello cada cambio anticipado está contenido en un módulo de no debe impactar en otros módulos.

- **Protección** Cada módulo puede ser probado individualmente y depurado para asegurar que funciona correctamente. Pero si algo va mal dentro de un módulo, éste debe de contener el error y frustrar la propagación a través del programa (protegiendo el resto de los módulos de un error).

Programación Interactiva

En "Inventing on Principle", Bret Victor ((51) citado por Davis (9)) describe el proceso de la programación conocida como " editar (el código), compilarlo

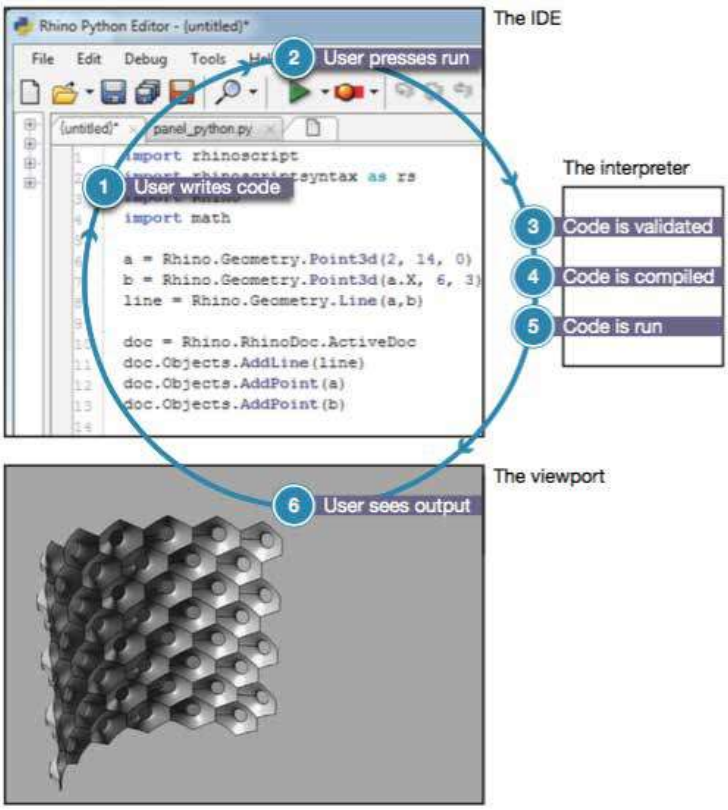


Figura 2.19: Edita-Compila-Corre. Procedimiento común entre los programadores en el cual se edita un código, se compila y se verifica el resultado, de esta forma se puede llegar a una solución. Este es el caso de un código en Python para el modelador **Rhino**, la mayoría de las veces este es un proceso que tarda segundos para ver el *output*, aunque a veces los códigos pueden ser muy demandantes computacionalmente (9).

y ejecutarlo, ver el resultado". Esta secuencia de eventos es comúnmente conocida como el ciclo *Edita-Compila-Corre*. En este ciclo, el programador *edita* el código, presiona un botón para activar el código, y luego espera. Primero se requiere tiempo a que la computadora valide el código, después para que lo *compile o traduzca*, usando el motor del lenguaje de programación, unas instrucciones legibles para la máquina, y finalmente la computadora *corra o ejecute* el conjunto de instrucciones. Hasta este punto el programador puede ver lo que el código produce. **El proceso interactivo.** El punto crucial de todos los ambientes de programación interactiva es remover la latencia entre la escritura del código y correr el código. Ambientes de programación interactiva logran esto con los siguientes conceptos:

- **Automatización** En lugar de esperar para que el usuario manualmente le diga al ciclo Edita-Compila-Corre que se ejecute, el ciclo se puede definir automáticamente y mostrar los resultados cuando el código cambia. Por simples cálculos, no es práctico re-compilar y re-calcular el proyecto completo cada vez que el código cambia, especialmente si los cambios tienen un pequeño impacto en el producto final.

- **Secuenciar** Así como el código cambia, las nuevas acciones son automáticamente ejecutadas en una secuencia "rítmica" o temporal mientras las acciones viejas son sin problemas descontinuadas ((47) citado por Davis (9)), lo cual evade el detener y reiniciar (ciclo Edita-Compila-Corre). Este método ha sido adaptado desde generar geometría simple hasta el tiempo medido en la música. Sin embargo, para los arquitectos que hacen demanda operaciones geométricas computacionalmente demandantes, no es tan importante generar la geometría rítmicamente sino rápidamente. Por esta razón, secuenciar es inadecuado para un contexto arquitectónico.
- **Hot-Swapping (Intercambio rápido)** En lugar de compilar cada linea de código, el hot-swapping permite compilar pequeños pedazos de código independientemente y después integrarlos con las partes sin cargar del programa - mientras que el programa en conjunto continúa corriendo. Esto reduce la latencia de compilación pero no reduce la latencia de correr el código.

Programación Visual Interactiva

El diseño involucra frecuentemente relaciones visuales, sistemáticas, o geométricas entre las partes que lo componen. La mayoría de las veces, estas relaciones son desarrolladas mediante flujos de trabajo que trasladan del concepto al resultado mediante un camino y el uso de reglas. Probablemente sin saberlo, nosotros estamos trabajando algorítmicamente, definiendo paso a paso un conjunto de acciones que siguen una lógica simple de entrada, procesamiento y salida. Programar nos permite continuar con el mismo trabajo que realizamos pero formalizando nuestro procedimiento, o algoritmo. Los algoritmos pueden generar cosas inesperadas, salvajes o increíbles... pero no son mágicos. De hecho son muy estrictos. Para explicarlo de manera sencilla se utilizará un ejemplo muy práctico, se explicará el proceso para crear una grulla japonesa mediante la técnica del origami[1].

¿Cuál sería el algoritmo? Sería el conjunto de pasos a seguir para lograr un objetivo, los cuales pueden ser representados de un par de formas - tanto textuales como gráficas. En un ambiente de programación textual como es conocida, se necesita de escribir código que será procesado por un compilador para traducirlo en un lenguaje que entienda la computadora. En el caso de hacer origami para lograr la forma de una grulla japonesa, el algoritmo textual sería el siguiente (33):

[1] Técnica de realizar figuras u objetos con hojas de papel doblándolas sucesivas veces.

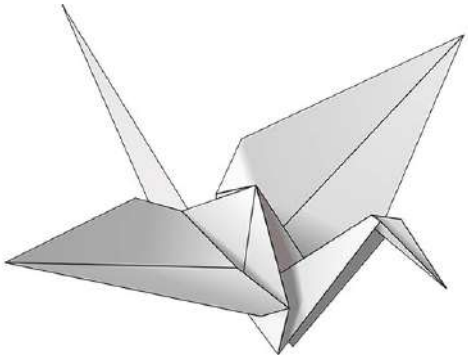


Figura 2.20: Grulla Japonesa. Origami objetivo realizado utilizando diferentes conceptos de programación (33).

Instrucciones textuales:

1. Empezar con una hoja de papel cuadrada, con color en la parte superior. Doblar a la mitad diagonalmente y abrir. Después doblar en el otro sentido.
2. Dar la vuelta al papel en la cara blanca. Doblar el papel a la mitad, realice bien el pliegue y abra, luego vuelva doblar en el otro sentido.
3. Usando los pliegues realizados antes, traer las 3 esquinas superiores a la esquina inferior. Aplane el modelo.
4. Doble las solapas triangulares en el centro y desdoble.
5. Doble la parte superior del modelo hacia abajo, haga bien el pliegue y desdoble.
6. Abra el doblez superior del modelo, trayendo y presionando los lados hacia adentro al mismo tiempo. Aplane hacia abajo, pliegue bien.
7. De vuelta al modelo y repita los pasos $4 - 6$ en el otro lado.
8. Doble las solapas superiores hacia el centro.
9. Repita en el otro lado.
10. Doble ambas “patas” del modelo , pliegue bien, después desdoble.
11. Doble las “patas” al reverso por dentro a lo largo del pliegue realizado antes.
12. Doble un lado al reverso y por dentro para crear la cabeza, después doble hacia abajo las alas.
13. Ahora ya tienes una grulla japonesa.

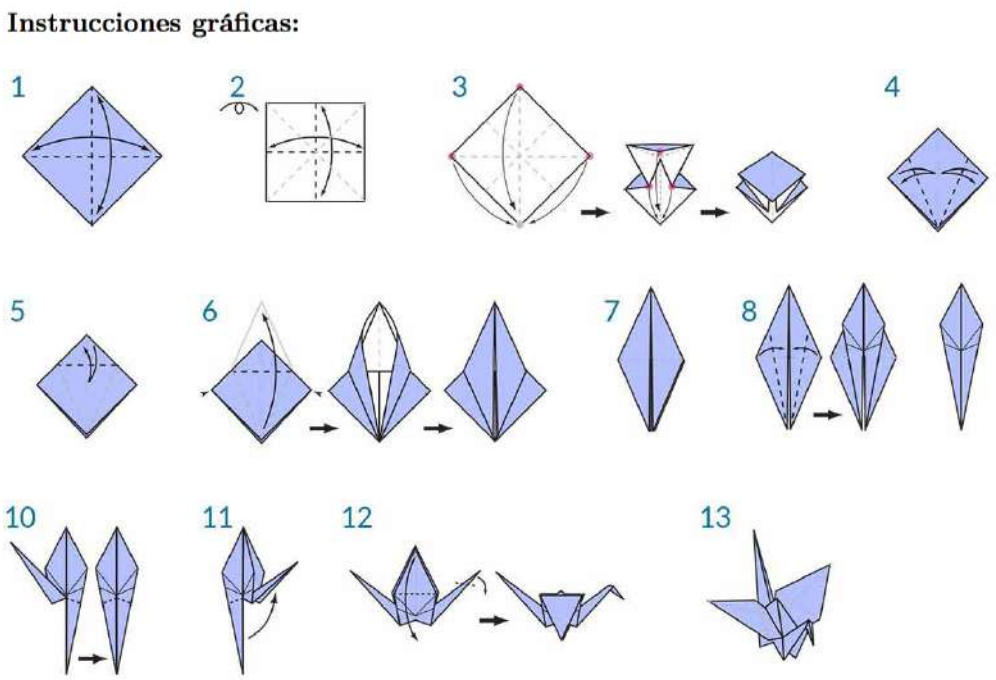


Figura 2.21: Programación Visual - Grulla Japonesa. Ejemplo de programación visual en origami (33).

Si en lugar de utilizar un procedimiento textual como el mostrado mediante la lista, se utilizara un algoritmo gráfico, una opción sería utilizar figuras explicativas como la mostrada en la imagen 2.21. Usando cualquier tipo de conjunto de instrucciones, el resultado es una grulla japonesa, y si alguien lo sigue tal cual entonces estás aplicando un algoritmo. La única diferencia es la forma en que se lee la formalización del conjunto de instrucciones, y esto nos conduce a *programar*. La programación, frecuentemente acortado el término de *Programación Computacional*, es el acto de formalizar el proceso de una serie de acciones en la cual un programa es ejecutado. Si utilizamos algún conjunto de instrucciones mostrado antes para obtener una grulla, utilizando una computadora y ejecutando el programa, entonces nos encontramos programando. La clave y el primer obstáculo que encontramos en la programación, es que debemos de confiar en alguna forma de abstracción para comunicarnos efectivamente con nuestra computadora. Esto nos da la opción de escoger el lenguaje que nos permita desarrollar nuestra aplicación, y hay una amplia variedad de éstos como por ejemplo *Javascript*, *Python* o *C*. Si podemos escribir un conjunto de instrucciones, como para el ejemplo de origami, solo necesitamos traducirlo a la computadora. En estos tiempos ya estamos en camino de que una computadora sea capaz de crear una grulla o incluso una multitud de diferentes grullas donde cada una tenga una variación específica. Con el

poder de la programación, la computadora podrá ejecutar cualquier tarea repetidamente, o un conjunto de tareas, que nosotros le asignemos sin el retraso o error humano. ¿Cuál es el tipo de programación que utilizarías para generar la grulla japonesa? ¿Gráficos, texto o una combinación? Si se piensa en una solución que involucre gráficos, *el lenguaje de programación visual* es altamente recomendado. Se utiliza el mismo marco de formalización; sin embargo, se definen las instrucciones de nuestro programa mediante relaciones a través de una interfaz gráfica. En lugar de escribir texto mediante una sintáxis, se conectan *"nodos"* con funciones pre-cargadas. La característica visual de la programación disminuye la barrera en la cual los diseñadores pueden transmitir sus ideas(33).

El funcionamiento de la programación visual se basa en el *"Data Flow Computing"* que se explica a detalle en la sección 2.2.1, por ello es recomendable revisarlo antes. Este tipo de programación tiene ciertas ventajas sobre su utilización frente a otros paradigmas de programación. En cuestión práctica, la programación visual es muy sencilla de aprender y la propuesta es acercarla a las áreas como Arquitectura o Ingeniería Civil para crear códigos visuales que permitan una optimización en procesos que se deben realizar cíclicamente o que son tediosos de realizar. En cuestión computacional, es muy interesante esta nueva metodología ya que en lugar de re-calcular cada operación geométrica, los gráficos con los que se construye el código, denominados *"Gráficos Acíclicos Dirigidos"* o simplemente *DAG* (por sus siglas en inglés) vistos en las figuras 2.16 y 2.18, son actualizados de acuerdo a un criterio definido por el procesador de la computadora, de manera similar a las celdas de las hojas de cálculo. Entonces cuando un nodo tiene una modificación en algún parámetro, el modelo se actualiza propagando los cambios a lo largo de la red para actualizar todos los nodos conectados posteriores. La actualización selectiva de los *DAG* guarda el cálculo geométrico que ya no son necesarios para la actualización, comprime en gran medida el tiempo entre escribir y correr el código.

Leitão, Santos y Lopes (citados por Davis (9)) concluyen, *"aprender programación textual toma más tiempo y esfuerzo que aprender un lenguaje de programación visual, pero este esfuerzo se recupera rápidamente cuando la complejidad de los problemas se convierte suficientemente grande"*.

Una cuestión importante de la programación visual, es que la activación de cada función depende de la conexión de redes *DAG*, cada vez que a un nodo se le conecta sus parámetros, este realiza su objetivo. Por tanto la transmisión de datos es la esencia que tiene esta metodología de programación, es decir que una buena gestión de la información permitirá tener un código limpio y funcional, además que reduce errores y memoria necesaria. En la siguiente sección se hablará más al respecto de esto.

Manejo de Información

El fundamento principal en la parametrización geométrica y de procesos es basada en la correcta utilización de la información, así como el manejo de ésta. En el contexto de la generación de modelos de edificaciones, la generación de estructuras es inicialmente basada en la geometría comprendida por sólidos, que describen la conceptualización de dicho edificio y que pueden ser fácilmente parametrizables, sin embargo se necesita de otros objetos como lo son nodos, líneas o superficies que le permiten al módulo desarrollado extraer información necesaria para definir las propiedades estructurales necesarias. En cuestión del cálculo numérico, el análisis estructural va a depender de información geométrica y de otros datos definidos por el usuario, como puede ser el tipo de material utilizado en los elementos estructurales, las secciones transversales de las estructuras, las solicitaciones de cargas impuestas o restricciones que desee colocar, siendo éstos parámetros no geométricos, por lo que es indispensable definir las correctas "conexiones" entre los objetos que reciben información y los que exportan. La clave está en saber qué información es útil para resolver el problema, qué datos son sensibles al cambio y qué variables pueden ser parametrizables, además de identificar cuáles otros datos no son necesarios y pudieran ralentizar el proceso.

Davis, durante su investigación (9), analizó 2002 modelos paramétricos de diseñadores de todo el mundo que fueron compartidos públicamente en el foro de Grasshopper (software de parametrización). Se revisaron 93 530 nodos DAG contenidos en los modelos y se hizo una categorización de acuerdo a su uso. Se observó que los nodos más comunes son los denominados "Number Slider", los cuales permiten al usuario ingresar valores numéricos. De hecho, sus conclusiones son que solo seis de los nodos más populares usados en los modelos son operaciones geométricas mientras que los demás son concentrados en el manejo de información, siendo que Grasshopper es una plataforma de modelación paramétrica especializada en geometría. Esto demuestra que en el modelado paramétrico, gran parte del trabajo realizado es el ingreso de datos, así como el manejo y gestión de la información necesaria para tener como resultado un modelo geométrico. La tabla 2.1 muestra la clasificación de los diez primeros nodos más comunes utilizados por los diseñadores para la generación de modelos geométricos.

Tabla 2.1: Resultados del análisis realizado por Davis (9), se muestran los diez nodos más comúnes en los 2002 modelos verificados. Estos nodos representan el 40 % de los 93 530 nodos existentes.

Clasificación	Nombre	Función
1	Number Slider	Selección de valor numérico
2	Panel	Lectura/Escritura de texto
3	List Item	Selecciona un objeto de listas
4	Point	Importa un punto
5	Curve	Importa una curva
6	Line	Importa un línea
7	Move	Mover geometría
8	Scribble	Dibuja en gráfico
9	Series	Crea series de números
10	Point XYZ	Crea un punto

2.2.2. Eficiencia de los Modelos Paramétricos

De acuerdo con Davis (9), saber cual es la eficiencia de un modelo paramétrico o la flexibilidad de uso es una tarea desafiante ya que no hay una definición propia de flexibilidad en la parametrización. Y por el hecho de que la flexibilidad depende de la situaciones que suceden en un proyecto, ésta se mantiene enigmática. Por ello la mejor técnica que se puede realizar, es la que presenta Davis en su tesis de doctorado, en la cual eligió modelos experimentales que tuvieran una mayor dificultad, no por el tamaño del modelo sino por la complejidad de algunos aspectos técnicos, que pueden ocurrir durante cualquier proyecto real. En el libro "Software flexible y seguro" de Henrik Christensen (2010), argumenta lo siguiente: "Cualquier software puede ser modificado durante su programación, en casos extremos se desecha todo el código y se escribe nuevamente desde cero". Durante la realización de este proyecto ocurrió una situación similar, se tenía una metodología para la estructuración de edificaciones rectangulares mediante un código implementado en un modelador paramétrico (en las secciones siguientes se explicará con más detalle), aún cuando trabajaba adecuadamente, la metodología empleada no permitía su expansión a edificaciones con otra geometría, por lo que se tuvo que desechar un código con un aproximado de 700 líneas por otro que permitiera una robustez mayor. Por lo que una buena referencia de medición es la que propone el "Compendio de estándares de calidad de software y mediciones" de Lincke y Welf (2007) (16), y el ISO/IEC 9126 para la calidad de los productos de software (55). Davis propone seis de las mediciones *cuantitativas*.

- **Tiempo de construcción.** El tiempo de construcción mide el tiempo que se toma construir un modelo desde cero. Claramente existen beneficios al tener un tiempo de construcción corto, particularmente si el modelo tiene reconstrucciones frecuentes durante un proyecto. Es usual que diferentes usuarios tengan distintos tiempos de construcción debido a que la familiarización del usuario con el ambiente de modelado ayuda a determinar que tan rápido éste puede construir el modelo. En general, el

tiempo de construcción para un modelo paramétrico es a menudo mayor que para otros métodos de modelado porque el proceso de crear parámetros y definir funciones explícitas requiere un grado de "front-loading", el cual es a menudo balanceado a través de tiempos de modificación.

- **Tiempo de modificación.** El tiempo de modificación mide el tiempo que toma hacer cambios a los resultados del modelo desde una instancia a otra. Tiempos de modificación cortos permiten a los diseñadores hacer cambios rápido, lo cual es una de las principales razones para usar un modelo paramétrico. Los cambios incluyen modificar el valor de los parámetros del modelo y esto puede incluir generalmente un proceso más arduo que es modificar las funciones explícitas. Cuando los diseñadores hablan de intentar "anticipar flexibilidad", ellos están refiriéndose a reducir el tiempo de modificación arreglando el modelo de manera que los cambios o manipulaciones se den a partir de los parámetros en lugar de las funciones (lo cual es mas lento para el segundo caso). Un punto importante aquí es que el tiempo de modificación es altamente dependiente de la organización del modelo, y particularmente cómo es impactado por la acumulación de cambios.

- **Latencia.** Es el periodo de tiempo que el usuario espera - después de hacer un cambio - para ver el último resultado o actualización del modelo. La latencia es causada por la computadora al realizar los cálculos necesarios para generar los resultados del modelo. Depende del hardware disponible, de la optimización del código y de los "cuellos de botella" de las funciones en los ambientes de modelado. A menudo esos cálculos resultan imperceptibles, pero modelos computacionalmente intensos la latencia puede durar minutos e incluso horas. La latencia es importante porque los diseñadores algunas veces fallan al observar cambios en un modelo particularmente si existe una pausa entre hacer un cambio y cuando se hace visible. Para que un modelo se sienta interactivo, investigaciones sugieren que la latencia debería ser menor de una décima de segundo y ciertamente no mucho más que un segundo. En muchos casos esto es imposible dada la demanda de varios cálculos geométricos, las limitaciones del hardware y los cuellos de botella en el interior de los algoritmos los ambientes de modelado.

- **Dimensionalidad.** La dimensionalidad es el número total de parámetros de un modelo. Visto de otro modo, el número de dimensiones del espacio de búsqueda en un modelo. Entonces, un modelo paramétrico ideal debe ser el que abarque todas las variaciones que el diseñador desee explorar con la menor dimensionalidad.

- **Tamaño.** Este parámetro se simplifica usando la medición de *"Líneas de Código (LOC por sus siglas en inglés)"*, aunque no es muy precisa

esta medición, ésta relaciona la complejidad del código con el número de errores posibles. En los diagramas DAG, se considera un módulo como una línea de código.

- **Complejidad ciclomática.** Es una medida utilizada primero en Ingeniería de Software. Se refiere al número de caminos independientes a través de un DAG. La complejidad ciclomática se calculó primero con la fórmula original de McCabe (1976) en la ecuación 2.1 ((29) citado por Davis (9)).

$$CC(G) = e - n + 2p \tag{2.1}$$

Donde:

- G: La gráfica.
- e: Número de extremidades. Siendo una sola extremidad las conexiones paralelas.
- n: Número de nodos. Sin contar nodos no funcionales (como los de texto).
- p: Número de gráficas independientes (partes).

Henderson-Seller y Tegarden (1994) (21) proponen una modificación para poder usarla en modelos paramétricos con más de una entrada y una salida. Dicha expresión se muestra en la ecuación 4.1. Un ejemplo de su aplicación se indica en la figura 2.22.

$$CC(G) = (e + (i - 1) + (u - 1)) - n + 2 \tag{2.2}$$

Donde:

- G: La gráfica.
- e: Número de conectores. Siendo un solo conector las relaciones paralelas.
- n: Número de nodos. Sin contar nodos no funcionales (como los de texto).
- i: Número de entradas (inputs).
- u: Número de salidas (outputs).

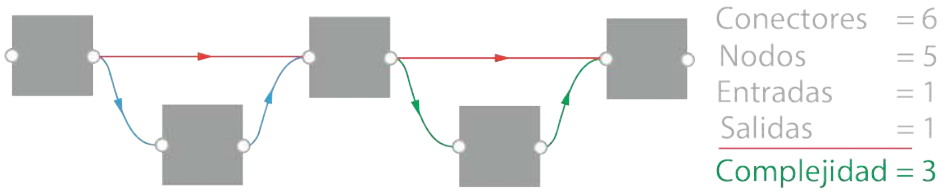


Figura 2.22: Complejidad Ciclomática. Cálculo de la complejidad ciclomática en un ejemplo de programación visual de un modelo paramétrico que utiliza redes *DAG*. Figura adaptada de (9)

Mientras sea posible comprender los caminos de los gráficos *DAG* interactúan se tendrá la idea de complejidad, pero esto se vuelve cada vez más difícil mientras las conexiones aumentan. Como resultado, McCabe recomienda re-estructurar cualquier código con una complejidad ciclomática mayor a 10. ((52) citado por Davis (9)).

Además de dichas mediciones cuantitativas, Davis (9) menciona algunas mediciones *cualitativas* relacionadas con el modelado paramétrico:

- **Exactitud.** Capacidad del programa para realizar la tarea que se le asigno. Aunque puede parecer trivial, en ocasiones es complicado debido a la variabilidad de los parámetros.
- **Robustez.** Reacción apropiada para condiciones de uso anormales.
- **Reutilización.** Habilidad para usar elementos del software en diferentes aplicaciones. La reutilización de modelos paramétricos también es una preocupación de muchos arquitectos.
- **Eficiencia.** Habilidad para usar las menores demandas posibles en los recursos del hardware disponible. Esto es pertinente porque la eficiencia de un programa determina la latencia. En casos extremos la eficiencia de un modelo puede determina si es viable, ya que ciertos cálculos geométricos son computacionalmente demandantes, tanto que la ineficiencia puede hacer más lento la generación de modelos al punto de volverlos imprácticos. Meyer (1997) menciona "la optimización extrema puede hacer que el software sea tan especializado que puede no encajar para hacer cambios y reusarse" (31) (citado por Davis (9)). La eficiencia puede ser importante pero necesita ser balanceada con otros atributos como la exactitud y la reutilización.
- **Portabilidad**: La facilidad de transferir productos de software a varios sistemas o ambientes.
- **Facilidad de uso.** La facilidad con la cual personas con diferentes antecedentes y cualidades pueden aprender a usar el software y aplicarlo para resolver problemas. Para los arquitectos la facilidad de uso aplica tanto

para el ambiente de modelado como para el modelo. Un ambiente de modelado sencillo implica el flujo de trabajo en la interfaz del usuario y el modelado. Un diseñador que le sea familiar el ambiente de modelado, así como la aplicación tenga cierta facilidad de uso, impactará directamente con el tiempo de construcción y modificación.

- **Funcionalidad.** Capacidades provistas por la plataforma de modelado, es decir, los tipos de geometría permisible, los tipos de relaciones y el método para usarlas.

De acuerdo a lo anterior podemos concluir que para garantizar la eficiencia y/o flexibilidad de un modelo paramétrico se deben tener ciertas consideraciones con respecto a las mediciones cuantitativas y cualitativas. Sin embargo como podemos inferir, es prácticamente imposible considerar "al pie de la letra" cada una de éstas, debido a que muchas de éstas se encuentran estrechamente relacionadas y por ende pueden conflictuar.

2.3. Dynamo: Programación Visual para el Diseño

El diseño computacional se refiere a la habilidad de ligar la solución de problemas con poderosos y novedosos algoritmos que hacen la función de automatizar, simular, codificar, parametrizar y generar diseños. En años recientes, el diseño computacional ha tenido mucho impacto en áreas como lo es la arquitectura y el diseño en general. Prácticas de diseño, grandes y pequeñas, empezaron a invertir en capacidades computacionales que les permiten personalizar sus procesos y lograr nuevos alcances. La computación impulsó una variedad de tareas como lo es automatizar una producción que era redundante o muy tediosa. Al igual que en las áreas que utilizan la computación para realizar procesos de producción óptima, los diseñadores necesitaron marcos de referencia similares para construir herramientas propias que les ayudaran con el tipo de trabajo que desarrollan. Esta sección muestra conceptos importantes utilizados en el software de modelado paramétrico elegido **Dynamo**, y para ello se utilizó como referencia principal la página en línea que da una explicación completa del funcionamiento así como tutoriales para empezar a usarlo, **Dynamo Primer** (33).

Figura 2.23: Dynamo. Logotipo de la versión "Open Source" de **Dynamo**, software de programación visual (33)

2.3.1. ¿ Qué es Dynamo?

*"**Dynamo** es literalmente todo lo que creas"*, esto es lo que dicen en **Dynamo Primer** (33), la página en línea que permite conocer todos los aspectos sobre la utilización de este software de modelación paramétrico. Piensan esto debido a que puedes trabajar con este software utilizando la aplicación, tanto vinculada con otros software de *Autodesk* o en versión aislada, activando procesos de programación visual, o participando en la gran comunidad de usuarios y desarrolladores.

La Aplicación

Dynamo, la aplicación, es un software que puede ser descargado y ejecutado en la versión de modo aislado o como complemento de otro software como **Revit** o **Maya** que son parte de la familia de software de la compañía especializada *Autodesk*. Ellos lo describen de esta manera: *"**Dynamo** es una herramienta de programación visual que pretende ser accesible para programadores y no-programadores por igual. Brinda a los usuarios la posibilidad de codificar gráficamente algún comportamiento, definir piezas personalizadas con lógica, y usar codificación mediante algunos lenguajes de programación textual."*

El Proceso

Una vez que ya tenemos instalado **Dynamo**, el software nos permite trabajar con un proceso de programación visual donde se puede conectar elementos para definir relaciones y secuencias de acciones que componen algoritmos personalizados. Es posible utilizar estos algoritmos para una amplia gama de aplicaciones, desde procesamiento de datos hasta generar geometría, todo esto en tiempo real sin necesidad de escribir ni un trozo de código.

La Comunidad

Dynamo no sería lo que es sin el grupo tan fuerte de usuario ávidos y activos desarrolladores por el número de contribuciones. Integrando la comunidad mediante una página tipo *Blog*, agregando trabajo a la galería, o discutiendo en el foro. Este es un aspecto muy importante, ya que debido a que estamos hablando de un lenguaje de programación, no puede existir un manual que te explique como desarrollar una aplicación en particular. Entonces, una comunidad tipo foro ayuda a resolver dudas de sintaxis, lógicas e incluso puedes subir la conexión *DAG* donde tienes algún problema o duda y existen personas tan involucradas o incluso desarrolladores del propio sistema que te hacen recomendaciones y te añaden funciones al código para obtener el resultado deseado. Este foro fue muy útil durante la elaboración del trabajo, agradezco mucho a las personas que pudieron ayudarme desde que empecé a conocer

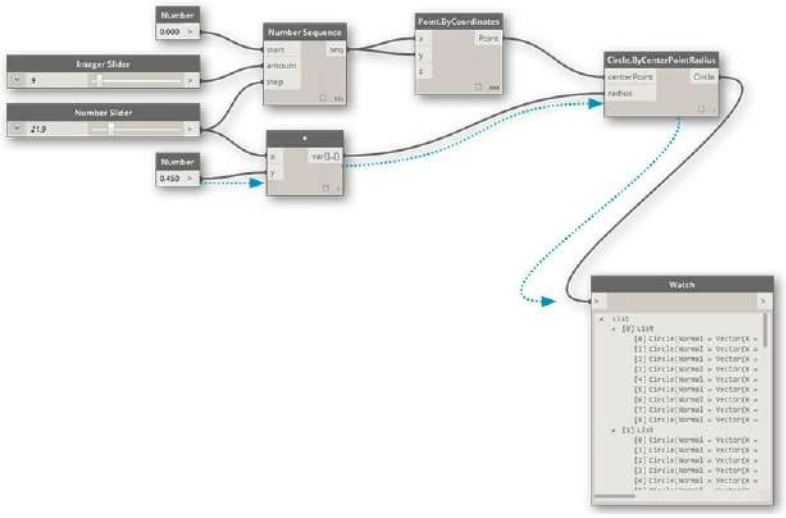


Figura 2.24: Dynamo. Utilización de red tipo *DAG* para realizar la conexión de información y ejecutar el código (33).

el funcionamiento e incluso ya cuando había dudas de un nivel más alto. De manera similar he tenido el gusto en participar resolviendo algunas dudas.

La Plataforma

Dynamo está planteado como una herramienta de programación visual para diseñadores, permitiéndonos hacer herramientas que usan librerías externas o cualquier producto de *Autodesk* que tenga un API. Con **Dynamo Studio** (versión *stand alone o aislada* de paga) se puede desarrollar programas en una aplicación de estilo *Sandbox*, y además el ecosistema de **Dynamo** sigue en crecimiento. El código fuente es *Open Source o Libre*, permitiendo extender las funcionalidades. El proyecto se puede obtener en *Github*[1] y buscar los trabajos en proceso de los usuarios que personalizan **Dynamo** . En la imagen 2.25 se observan algunos proyectos que usuarios de la comunidad en **Dynamo Primer** compartieron públicamente, para denotar el alcance que tiene esta herramienta de programación visual en diversos campos, ya que aparte de diseño en arquitectura también se puede aplicar en aplicaciones de matemáticas e ingeniería. Se añadió el nombre de usuario en cada imagen para hacer referencia y darle el crédito merecido.

[1] Plataforma en línea para el traspaso de códigos de programación

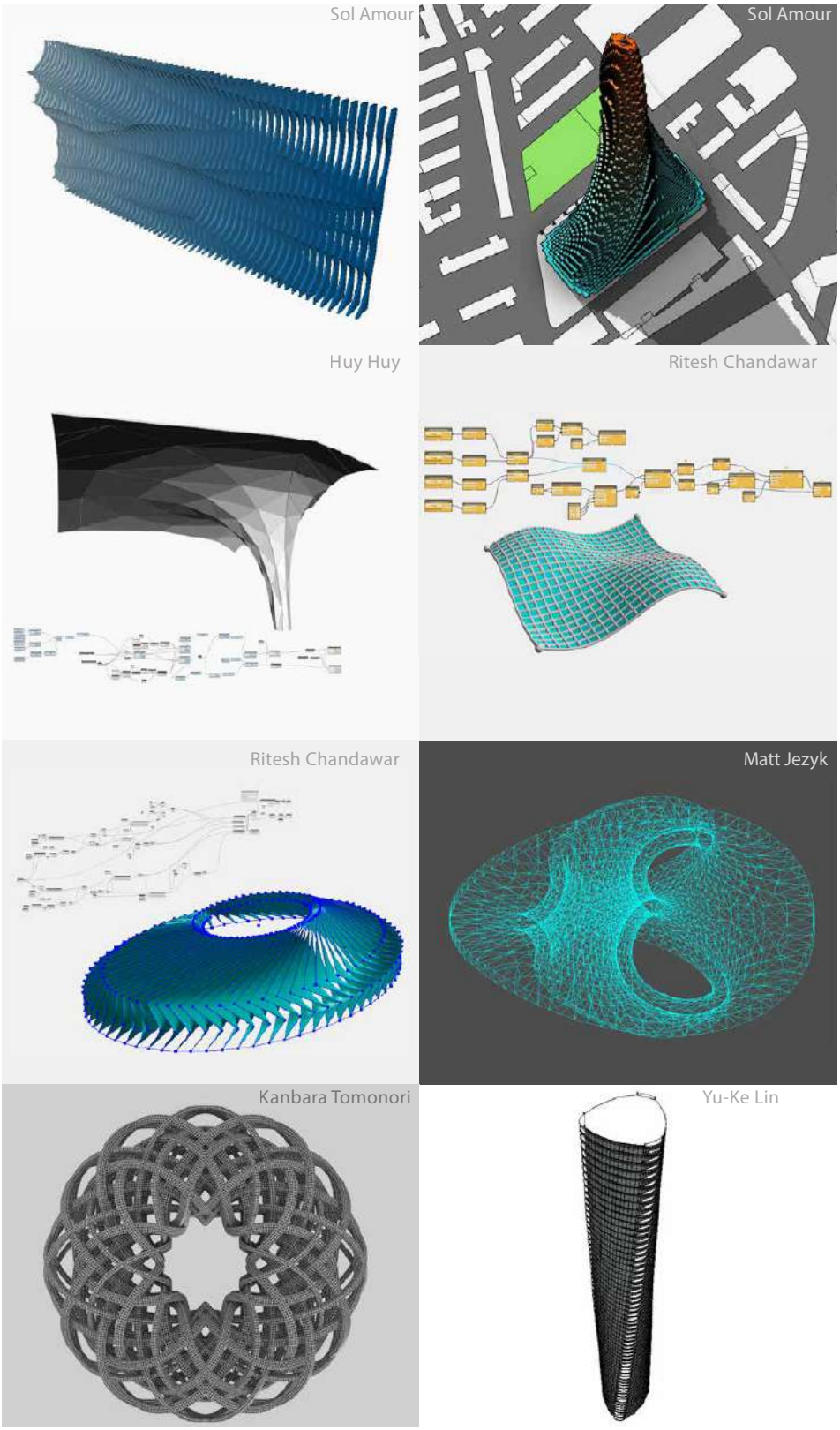


Figura 2.25: Dynamo. Galería de imágenes que los usuarios de **Dynamo** comparten públicamente al foro de **Dynamo Primer** (33).

2.3.2. Anatomía de un Lenguaje de Programación Visual

Lenguaje de Programación Visual es un concepto que provee a los diseñadores el significado de construir relaciones programadas utilizando interfaces gráficas. En lugar de escribir un código desde cero, el usuario tiene la capacidad de ensamblar relaciones personalizadas al conectar funciones "preempaquetadas" para crear un algoritmo personalizado. Esto significa que un diseñador puede usar conceptos de computación, sin la necesidad de escribir un código. **Dynamo** permite a los diseñadores realizar diseño computacional y procesos de automatización a través de una interfaz de programación visual basada en "nodos". Además, **Dynamo** brinda al diseñador la ventaja de utilizar los flujos de trabajo o *"workflows"* incorporados a un ambiente *BIM* o *Modelado de Información de la Construcción.* En esta sección se hablará de los componentes utilizados por un software basado en programación visual. Se hablará en específico del software **Dynamo** debido a que éste fue el seleccionado para realizar este trabajo, sin embargo entre los programas de este tipo se tiene un funcionamiento muy similar.

Dynamo, el cual tiene opción de utilizarse como un software de parametrización Open Source, es una interfaz de programación visual que conecta el diseño computacional con la modelación de información de construcción, o por sus siglas en inglés (BIM). Con Dynamo, los usuarios pueden crear scripts para construir, modificar e intercambiar información de acuerdo a sus necesidades. El software de modelado permite diseñar edificaciones orgánicas y estructuras óptimas, más rápido que con el planteamiento tradicional. Esto es porque podemos crear, asociar y analizar múltiples parámetros, podemos iterar y evaluar distintas opciones de diseño y construir estructuras mediante principios naturales y matemáticos (20). La forma de hacer esto es mediante la programación visual que se brinda en el espacio de trabajo, y se realiza mediante una conexión de nodos con cables para especificar el flujo lógico que seguirá el programa.

Nodos

En **Dynamo**, los *Nodos* son objetos que se conectan para formar un programa visual. Cada nodo hace una operación - algunas veces pueden ser tan simples como solo guardar algún valor numérico o algunas otras tan complejos como para generar toda una geometría solicitada. La mayoría de nodos utilizados en **Dynamo** están compuestos por cinco partes. Aunque existen algunos otros que tienen excepciones debido a la tarea que realizan. La figura 2.26 muestra las partes del nodo. Los elementos principales son:

1. Nombre de la función.

2. Cuerpo del nodo, el cual tiene diversas funciones al hacer click derecho.

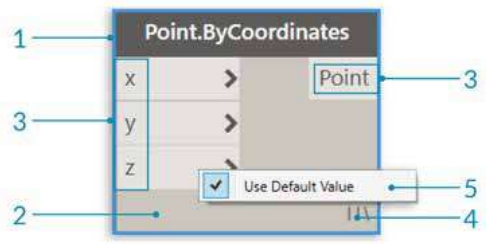


Figura 2.26: Dynamo. Anatomía de un nodo de **Dynamo** utilizado como base para lenguajes de programación visual (33).

3. Puertos de entrada y salida, son los receptores de información mediante parámetros, así como los resultados generados.

4. Ícono de opciones de distribución, utilizado para designar la forma de distribuir ciertas geometrías y operaciones

5. Valor por defecto, dando click derecho se puede especificar cuales valores no utilizar.

Cables

Los *Cables* (*Wires*) conectan las funciones nodo para crear relaciones y establecer el flujo del programa visual. Se puede pensar en ellos literalmente como *cables eléctricos* que transmiten electricidad de un objeto a otro. Los *Cables* conectan un puerto de salida de un nodo a un puerto de entrada de otro nodo. La dirección de estas conexiones establecen el *Flujo de Datos* en el programa visual (ver figura 2.28). Aunque los nodos se pueden organizar como deseemos en el espacio de trabajo o *canvas*, los puertos de salida están del lado derecho de cada nodo y los puertos de entrada están en el lado izquierdo, por eso generalmente el flujo del programa se mueve de izquierda a derecha.

Librerías

La *librería* de **Dynamo** contiene a los nodos que se añaden al espacio de trabajo para definir el programa visual que será ejecutado. Los nodos contenidos aquí tienen cierta jerarquía, los cuales incluyen nodos por defecto en la instalación, nodos personalizados que podemos crear y nodos de la plataforma en línea de librerías que se pueden añadir a **Dynamo**. Las librerías son parte fundamental para crear un programa visual, ya que con las funciones que se encuentran contenidas se realizan las conexiones mediante cables para implementar la aplicación a desarrollar. Algunos nodos comunes se muestran en la imagen 2.28.

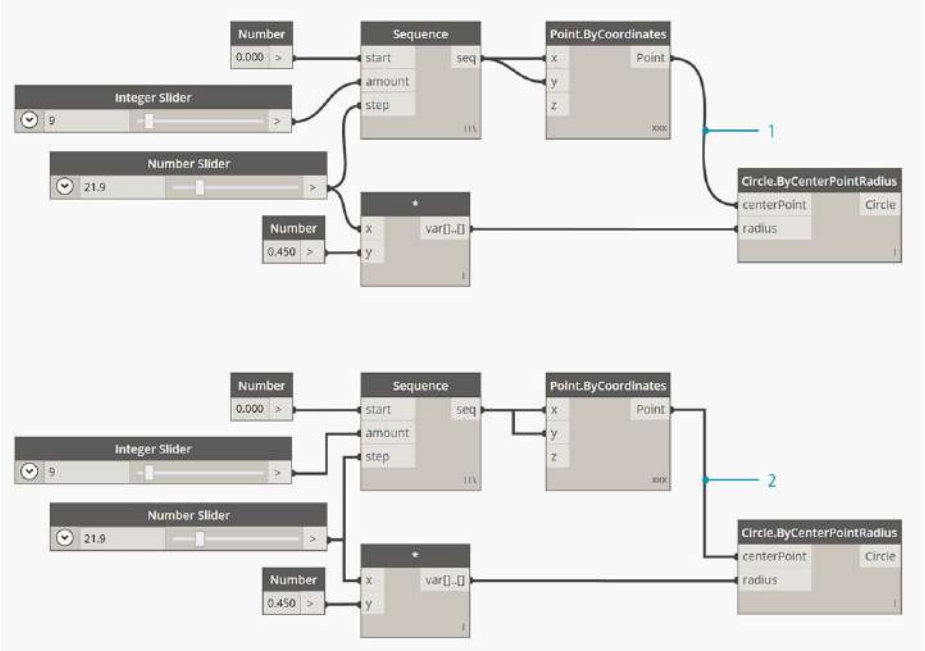


Figura 2.27: Dynamo. Flujo de datos que se define al conectar *cables* en los puertos de *nodos* del programa visual, se observan los dos tipos de cables que se pueden utilizar en **Dynamo** (33).

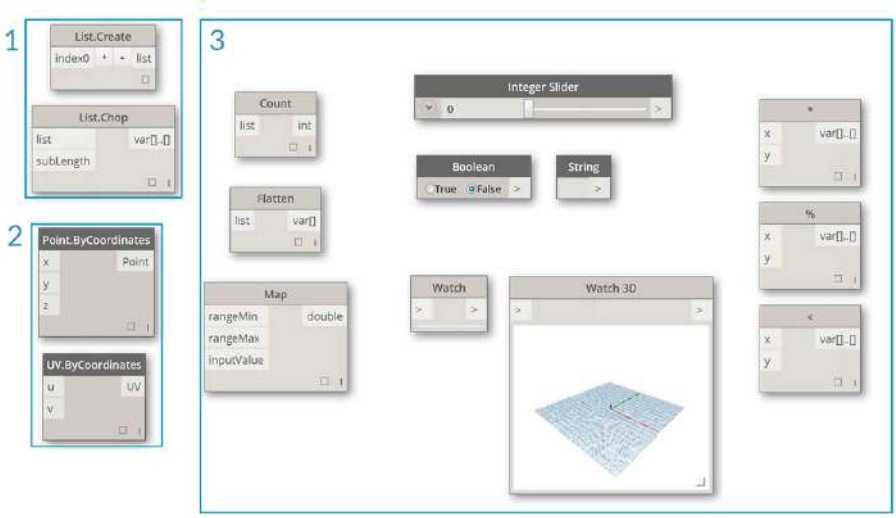


Figura 2.28: Dynamo. Nodos comunes encontrados en la librería estándar de **Dynamo** (33).

Figura 2.29: Dynamo. Representación del *Conejo de Stanford*, utilizado en pruebas experimentales para algoritmos relacionados con geometría (33).

2.3.3. Diseño Computacional Geométrico

Geometría es el lenguaje para el diseño. Cuando un lenguaje de programación o ambiente tiene un parte geométrica en su núcleo, se pueden desbloquear las posibilidades para diseñar precisamente robustos modelos, diseñar rutinas de automatización, y generar iteraciones con algoritmos. La *Geometría*, se define tradicionalmente como el estudio de la forma, el tamaño, la posición y propiedades del espacio. Este campo tiene una basta historia miles de años antes. Con la llegada y la popularización de la computadora, se ganó una herramienta poderosa en la definición, exploración, y generación de geometría. Ahora es fácil calcular el resultado de interacciones geométricas complejas. Entender la geometría en el contexto de algoritmos, computación y complejidad puede sonar intimidante; sin embargo, existen algunos principios que se establecen como fundamentos para empezar a construir aplicaciones avanzadas:

- La geometría es *información* - para la computadora y **Dynamo**, un conejo no es tan diferente que un número.
- La geometría depende en la *abstracción* - los elementos geométricos son descritos por números, relaciones y fórmulas en un espacio coordenado.
- La geometría tiene *jerarquía* - puntos se combinan para crear líneas, las líneas se combinan para crear superficies y así sucesivamente.
- La geometría simultáneamente describe tanto *la parte y el conjunto* - cuando se tiene una curva, es tanto la forma como todos los posibles puntos a lo largo.

Estos principios ayudan a tener en cuenta con qué estamos trabajando, y con ello poder componer, descomponer y recomponer con fluidez, geometrías mientras se crean modelos complejos. Ahora, tomemos en cuenta la relación entre la abstracción y la jerarquía de la geometría. Para empezar, se puede utilizar la dimensionalidad como un descriptor de la "cosa" que modelamos. El número de dimensiones requerido describe la forma y da una primer vista de como la geometría es organizada jerárquicamente.

Dimensionalidad es una forma conveniente para empezar a categorizar la geometría pero no es necesariamente la mejor. Después de todo, no solo se modela con *puntos, líneas, planos y cajas* - ¿Que pasaría si quiero algo curvo? Existe un otras formas de categorizar los tipos de geometría que son completamente abstractos, como por ejemplo propiedades como la orientación, el volumen, o la relación entre sus partes. En las secciones siguientes se presentarán las geometrías más utilizadas.

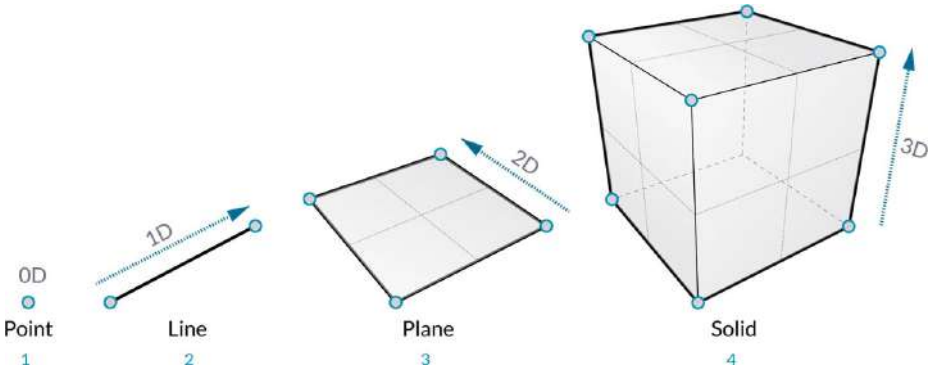


Figura 2.30: Dynamo. Representación del concepto de dimensionalidad jerárquica utilizando diferentes objetos geométricos (33).

Vectores, Planos y Sistemas Coordenados

Los *Vectores, Planos y Sistemas Coordenados* son el primer grupo de geometrías abstractas. Ayudan a definir ubicación, orientación y el contexto espacial para otras geometrías. Un *vector* es una cantidad geométrica que indica una dirección y magnitud, más que un elemento geométrico representan una cantidad y son esenciales porque son *"ayudantes"* para generar otras geometrías, por ello si se crea uno en el espacio geométrico no se verá nada. Un *plano* es un *"ayudante"* bidimensional que se extiende infinitamente, y son utilizados comúnmente para operaciones geométricas como el corte de figuras. Un *plano coordenado* define el origen de nuestras figuras, por lo que son utilizados para asignar ubicación a geometrías.

Puntos

Los puntos son el cimiento de otras figuras al ser creadas. Se necesitan al menos dos puntos para crear una curva, se necesitan al menos tres puntos para generar un polígono y así sucesivamente. Definen posición, orden, y relaciones entre otros puntos y permiten generar geometrías de órdenes mayores como círculos o curvas. Un punto es definido nadamas que por sus coordenadas espaciales.

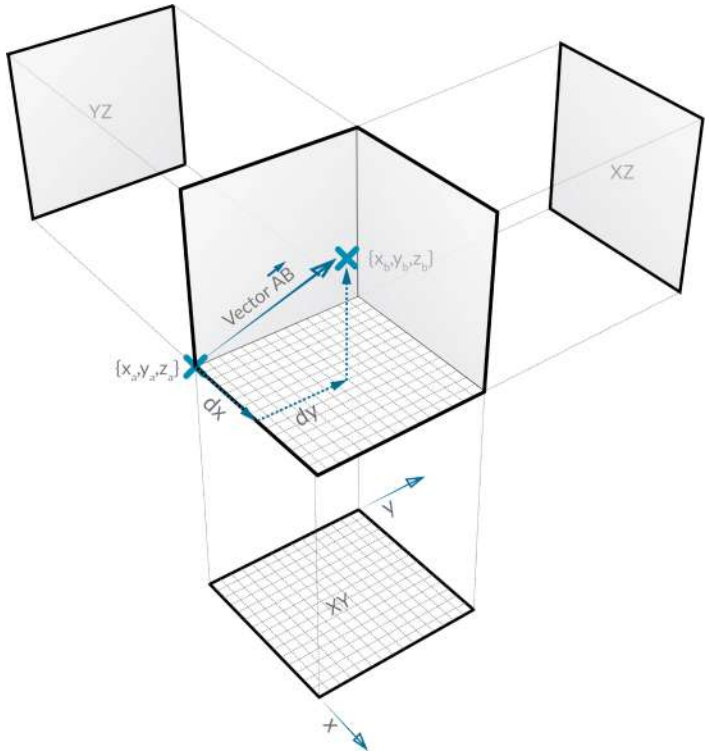


Figura 2.31: Geometría en Dynamo. Vectores, Planos y Sistemas Coordenados (33).

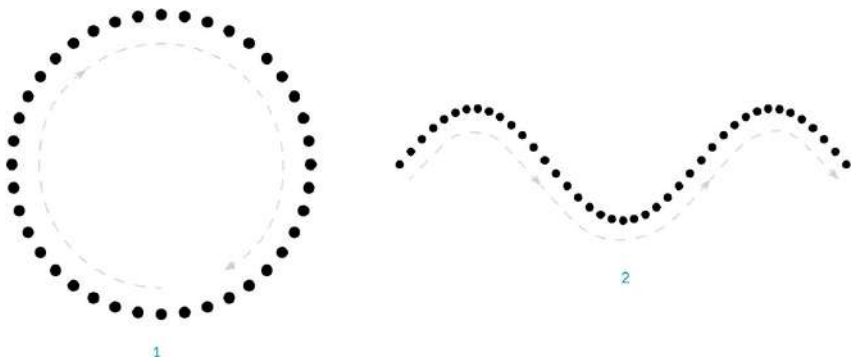


Figura 2.32: Geometría en Dynamo. Puntos como cimiento de geometrías, formando un círculo y una curva (33).

Curvas

El término de *curvas* se generaliza para un conjunto de geometrías con curvatura o incluso rectas. Técnicamente una curva describe cada posible punto que se puede encontrar en una función. No importa que tipo de curva es, siempre se podrá evaluar un parámetro en una posición. A pesar de la apariencia de la forma, todas las curvas tienen un punto inicial y uno final, lo cual ayuda a entender su direccionalidad. Entre las curvas que existen se encuentran las *líneas* que son el caso más simple ya que son rectas y solo necesitan de un punto inicial y uno final. Al conectar varias líneas se obtiene una *polilínea*. Si se añade más complejidad se pueden crear *arcos, círculos o elipses* o incluso las denominadas *Nurbs* (Non-Uniform Rational Basis Splines) que son representaciones matemáticas que pueden modelar con precisión cualquier forma, por ello son ampliamente utilizadas.

Superficies

Una superficie es una forma matemática que es definida por una función y dos parámetros. Se utilizan los valores *U y V* para describir el espacio correspondiente. Esto significa que se tiene más información geométrica cuando se trabaja con este tipo de geometrías. Al igual que las curvas pueden no ser planas siempre, las superficies también se encuentran en ambientes tridimensionales. Existen algunos tipos como lo son la unión de múltiples superficies que integran una *Polisuperficie*, o *Superficies Nurbs* que son similares a las *Curvas Nurbs* pero formando una retícula en dos sentidos dintintos.

Sólidos

Los sólidos consisten de una o más superficies que contienen volumen a través de fronteras cerradas que indican "afuera" o "adentro". A pesar de cuantas superficies contengan, deben formar un volumen cerrado para ser considerados como sólido. Éstos se pueden crear uniendo superficies o utilizando operaciones que a partir de curvas se aplique una función de "extrusión" para ser creados. Debido a que son geométricas más complejas tienen ciertas características especiales, como en cuestión de topología, que tienen tres tipos de elementos como lo son los vértices, lados y caras. Y entre las operaciones posibles destacan la intersección, la extracción, la remoción de partes y la unión con otros elementos.

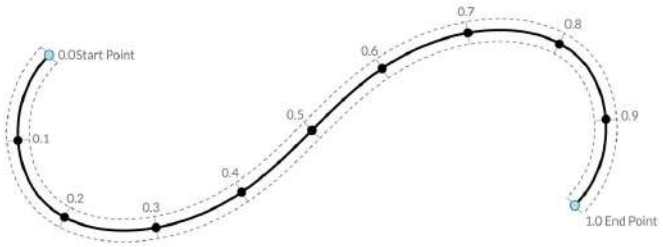


Figura 2.33: Geometría en Dynamo. Curva que muestra algunos puntos que la componen, además de su inicio y final (33).

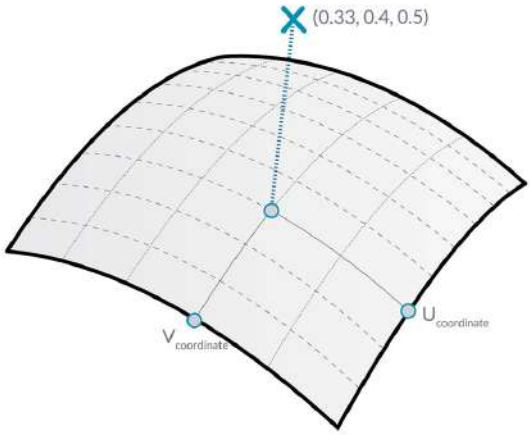


Figura 2.34: Geometría en Dynamo. Esquema de una superficie denotando los dos parámetros necesarios U y V (33).

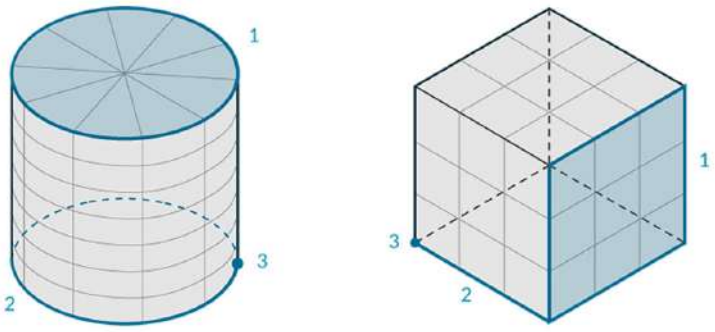


Figura 2.35: Geometría en Dynamo. Elementos que forman sólidos tridimensionales mediante caras de geometría de una dimensión menor (superficies)(33).

Distribución de Espacio Vertical. One World Trade Center localizado en la ciudad de Nueva York, cada vez se hace más necesario el uso de construcciones con distribución vertical (rascacielos o edificaciones tipo torre) por su eficiencia en cuanto al uso de terreno. (E. Muttio - 2016)

Capítulo 3

Programación de Módulos Paramétricos

3.1. Modelos de Edificaciones Paramétricas

El tipo de edificaciones que se pretende enfocar en este trabajo son las denominadas **"Edificaciones Torre"**, las cuales deben de cumplir ciertos requisitos para la simplificación en el modelo paramétrico utilizado. Las principales razones (necesarias para el primer procedimiento de estructuración) que se deben de considerar al desear estructurar un modelo tridimensional sólido son las siguientes:

1. El sólido tridimensional debe presentar forma prismática o cuasi-prismática regular donde se planee tener niveles o pisos que sigan un trayecto perpendicular desde la base.

2. Debido a la consideración de niveles completos y ascendentes con horizontalidad perfecta, se debe omitir niveles intermedios o "mezzanines".

3. Se recomienda que el sólido tridimensional debe describir la forma del exterior deseada en una edificación, omitiendo detalles arquitectónicos no necesarios para la estructuración temprana o concepción arquitectónica inicial.

4. El sólido no debe contener huecos en su interior, las plantas deben tener completa continuidad.

En esta sección se detallará el funcionamiento de una gráfica *DAG* que crea un prisma rectangular, el cual puede considerarse como una edificación. Después se hablará sobre el modelo estructural que se utilizó en todas las pruebas de los algoritmos para estructuración y análisis, la *Torre Mayor*.

3.1.1. Edificación Rectangular Mediante DAG

En esta sección se verá de manera básica como construir un modelo tridimensional utilizando el lenguaje de *programación visual* que proporciona **Dynamo**. El objetivo será generar un prisma rectangular utilizando las funciones integradas y conectando nodos para tener obtener un *DAG* funcional . Las siguientes imágenes muestran el procedimiento:

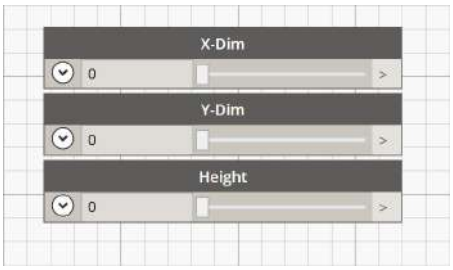


Figura 3.1: Procedimiento para generar un DAG. Esta primer imagen muestra los parámetros necesarios para generar un prisma rectangular. Se utilizaron las funciones *"Number Slider"*

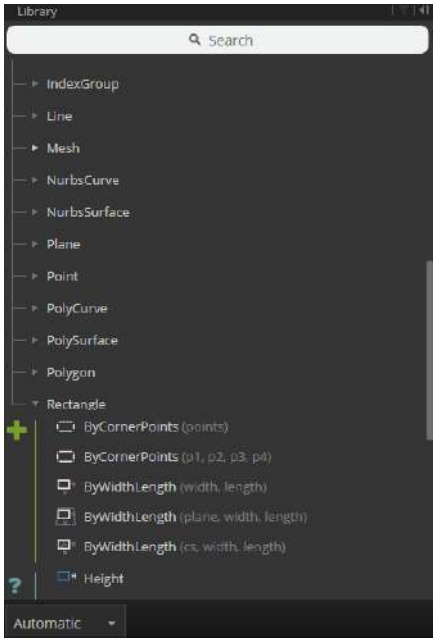


Figura 3.2: Procedimiento para generar un DAG. La segunda imagen muestra la biblioteca geométrica con la que cuenta **Dynamo** y las opciones para generar el rectángulo base.

Se selecciona la opción de generación de rectángulo mediante los parámetros *ancho* y *largo*. Aunque existen otras formas de generar un prisma rectangular, se tomo tomó la decisión de hacerlo de esta manera de forma que se pudieran

utilizar los parámetros mostrados en la figura 3.1. Para generar el prisma, se necesitarán de dos rectángulos, uno inferior y otro superior.

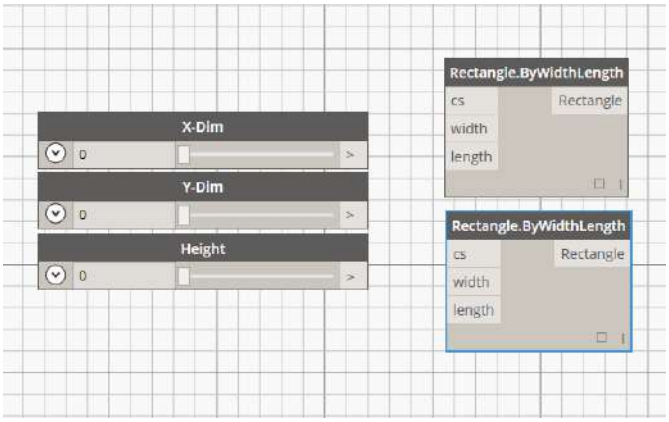


Figura 3.3: Procedimiento para generar un DAG. Se selecciona dos veces la función de la biblioteca *"Rectangle.ByWidthLength"*.

Cómo podemos ver en la imagen 3.3, además de los parámetros *ancho* y *largo*, la función requiere de un sistema coordenado que indique donde posicionar el rectángulo. Para ello, existe un objeto denominado *"Coordinate System"*, el cual se puede generar de diversas formas, para indicar su ubicación en el espacio tridimensional. Sin embargo, se utilizará un objeto preestablecido denominado *"Identity"*, el cual denota un objeto "Sistema coordenado" posicionado en las coordenadas $(0, 0, 0)$, es decir, en el origen. De esta manera reducimos conexiones en el DAG. Tal como podemos ver esta función no necesita un parámetro, entonces al solo colocarla sobre el espacio de funciones o *"Canvas"*, se ejecuta. Al conectar el sistema coordenado y los datos geométricos a la primer función *"Rectangle.ByWidthLength"*, se genera en el canvas el rectángulo que servirá como base para la edificación. La imagen muestra esta parte del procedimiento.

Para el siguiente paso se generará el rectángulo que se encuentra en la parte superior de lo que será el sólido. Para ello se utilizará la misma función *"Rectangle.ByWidthLength"*, pero a diferencia de la función de sistema coordinado utilizada antes, se tendrá que incorporar otra. Esto es debido a que dicha función regresa el origen de sistema coordenado y ahora buscamos que se encuentre desplazado en sentido vertical una magnitud igual a la elevación deseada para la edificación. Por lo que se empleó *"CoordinateSystem.ByOrigin"*, la cual genera un sistema coordenado ubicado en un punto geométrico de referencia. Solamente con asignarle el valor de altura o "Height" al parámetro Z es suficiente, ya que los otros dos valores los toma como cero. En la figura 3.6 se aprecian los dos sistemas coordenados y la función utilizada. Ahora lo que sigue es generar el rectángulo superior de la misma manera que se hizo para el inferior. Se conectan los parámetros necesarios al nodo del DAG y se obtiene el resultado mostrado en la figura 3.7.

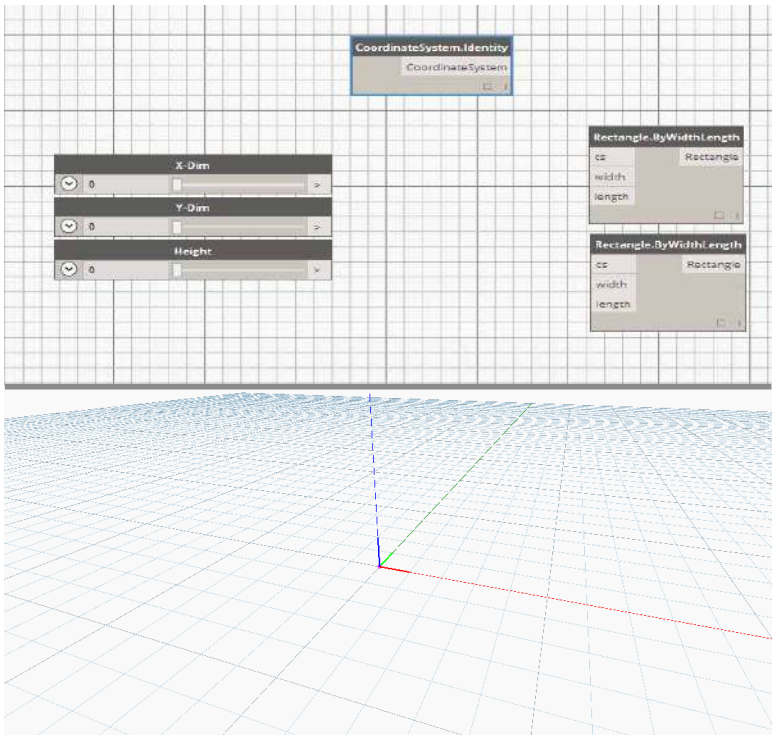


Figura 3.4: Procedimiento para generar un DAG. Se selecciona la función *"CoordinateSystem.Identity"* para indicar la posición del rectángulo base. Al colocarla en el *Canvas* se muestra en el espacio geométrico la indicación de que hay un objeto de tipo "Sistema Coordenado".

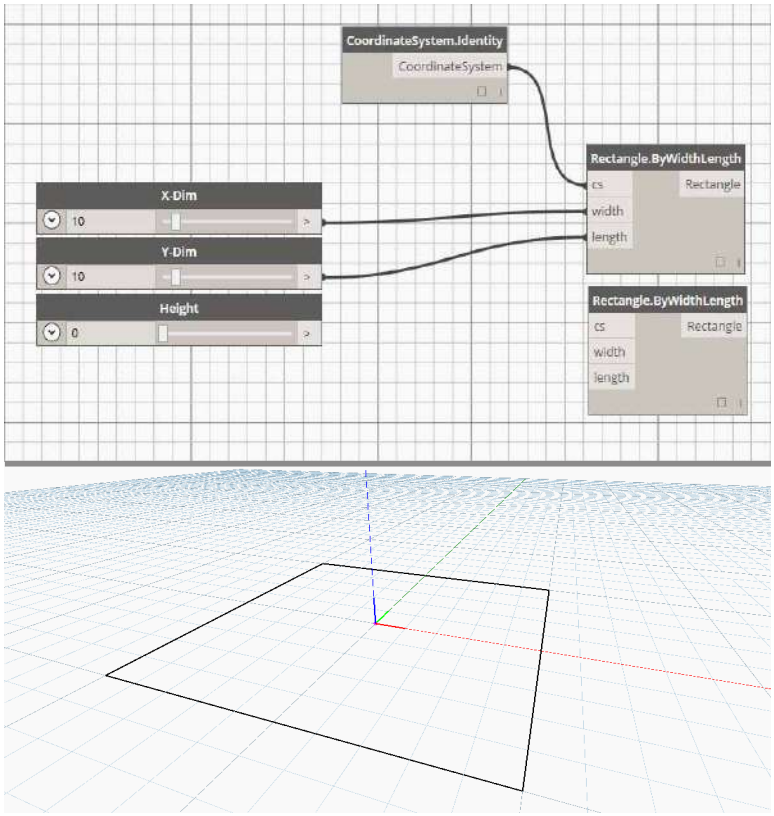


Figura 3.5: Procedimiento para generar un DAG. Conexión de la función *"CoordinateSystem.Identity"* con *"Rectangle.ByWidthLength"*, en el *Canvas* se muestra la generación del rectángulo base.

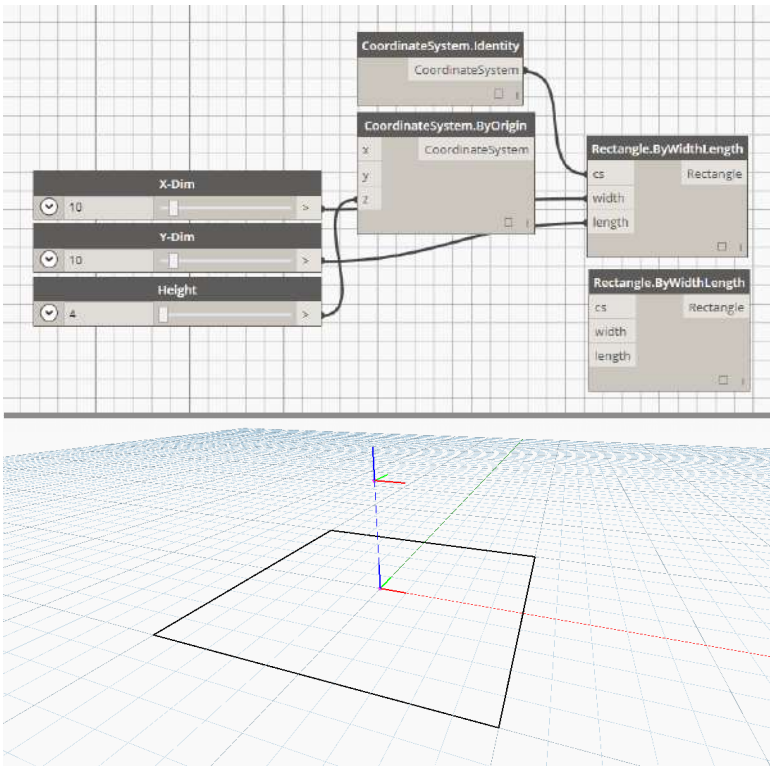


Figura 3.6: Procedimiento para generar un DAG. Se utiliza una nueva función para generar un objeto "sistema coordenado", la cual necesita de coordenadas de un punto para localizar su ubicación.

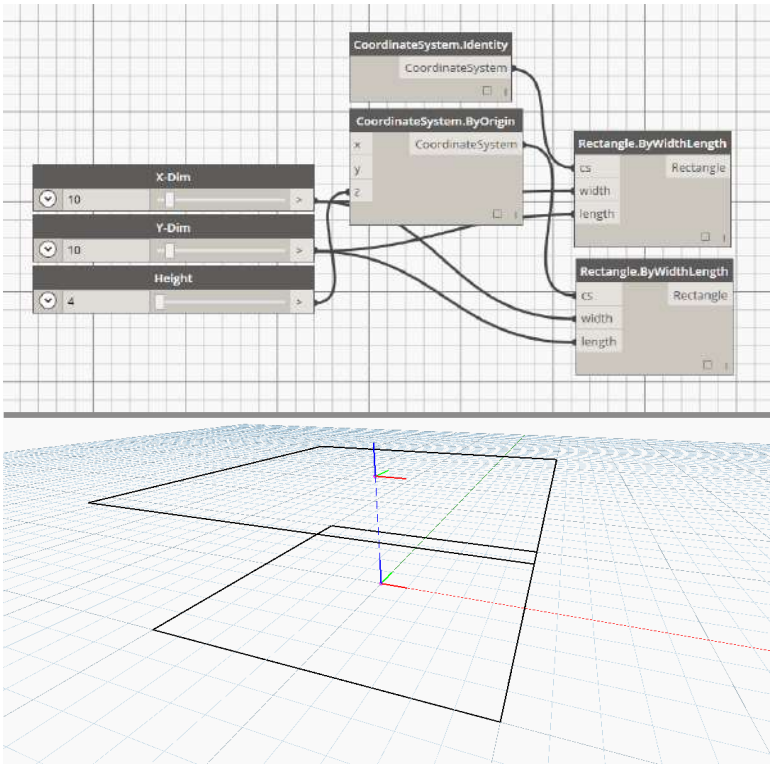


Figura 3.7: Procedimiento para generar un DAG. Generación del rectángulo superior mediante el uso de un sistema coordenado desplazado en el sentido vertical.

El motivo de generar los rectángulos anteriores es que se planea utilizar cierta función de **Dynamo** que permite crear un sólido a partir de una lista de superficies. La función se denomina "Solid.ByLoft", el algoritmo que tiene implementado permite reconocer un patrón entre las superficies y generar el contorno perpendicular o caras del sólido. La función se muestra en la imagen 3.8. En cambio la figura 3.10 muestra la conexión necesaria para generar el sólido, como podemos observar se utilizó una función intermedia entre las superficies y el sólido, dicha función se denomina *"List.Create"* y simplemente lo que hace es almacenar los datos proporcionados en una lista que el usuario va definiendo al hacer click en el botón integrado. De esta manera se pueden guardar cualquier tipo de variables o geometrías. La figura 3.10 muestra el resultado generado por este DAG.

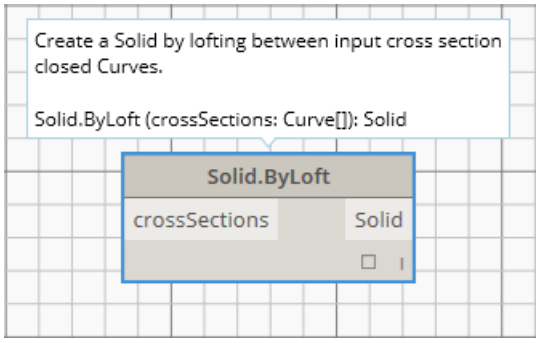


Figura 3.8: Procedimiento para generar un DAG. Función de **Dynamo** utilizada para generar un sólido mediante superficies.

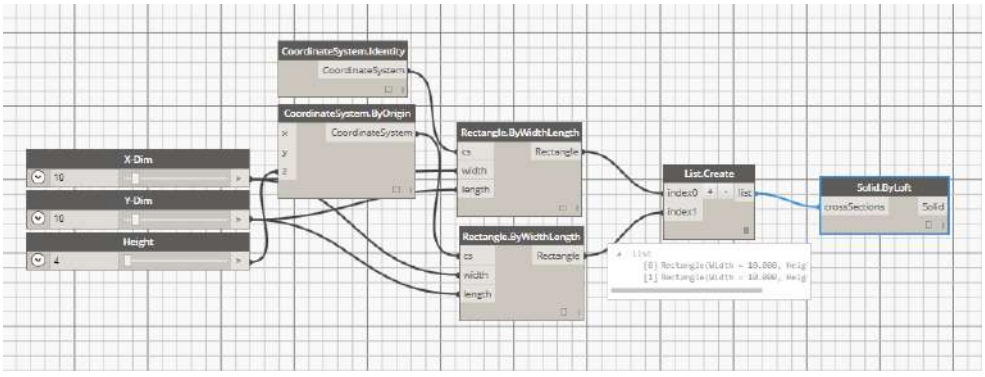


Figura 3.9: Procedimiento para generar un DAG. Red de conexiones paramétricas o DAG para generar el sólido prismático, se observa la utilización de la función *"List.Create"*.

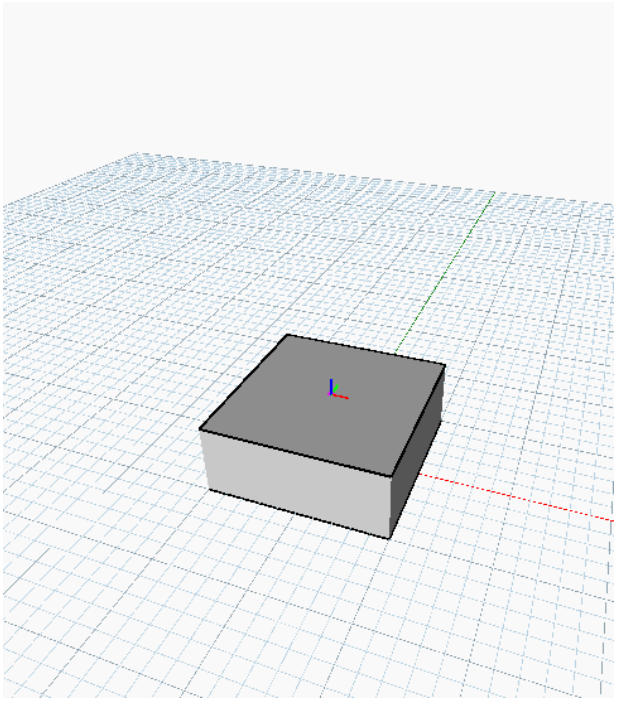


Figura 3.10: Procedimiento para generar un DAG. Sólido prismático generado por el DAG de la figura 3.9.

Quizá un usuario familiarizado con entornos *CAD* en donde la modelación se realiza mediante comandos y "clicks" del ratón, pueda pensar que generar un simple sólido utilizando todos estos pasos pudiera ser un tanto tedioso y "engorroso" es por ello que se puede errar al despreciar este método, lo cual es comprensible hasta el momento. Pero al tomar en cuenta que la conexión de información permite que los datos estén vinculados y al variar parámetros podemos obtener nuevas propuestas. Justamente se probará esto cambiando la altura del sólido mediante el parámetro "Height". La figura resultante se muestra en la imagen 3.11. La imagen 3.12 denota más modificaciones en ese mismo parámetro para denotar la facilidad de cambio en un modelo paramétrico. Es hasta este momento cuando se puede apreciar los beneficios de utilizar este sistema, y aunque pudiera pensarse que un cubo no tiene mucho relevancia en el cambio de dimensiones, hay que intentar extrapolar este comportamiento a otras aplicaciones más complejas donde los cambios tienen un costo temporal alto.

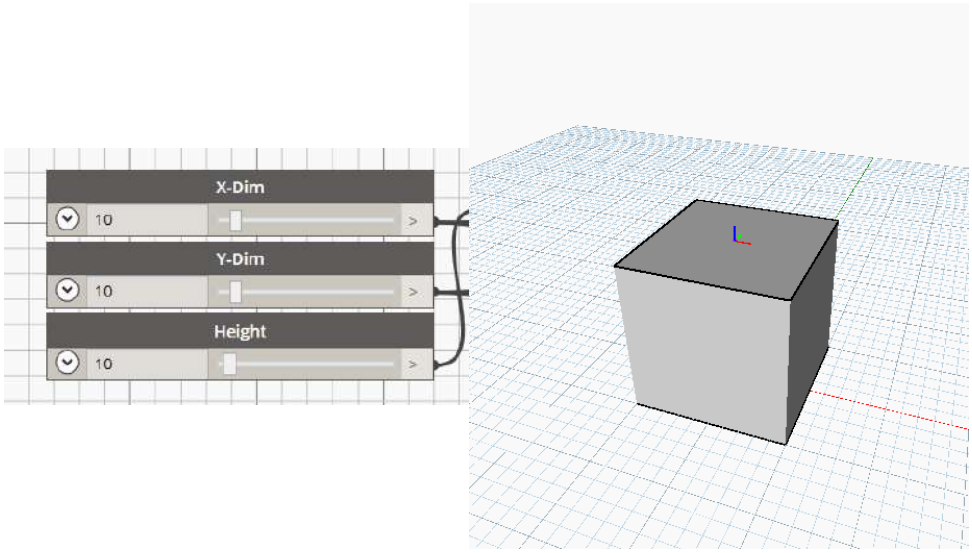


Figura 3.11: Procedimiento para generar un DAG. Modificación del parámetro "Height" para obtener un nuevo modelo prismático.

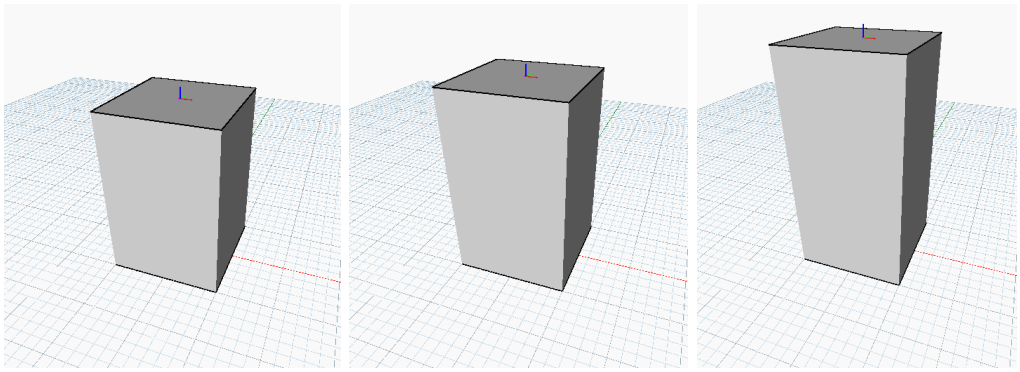


Figura 3.12: Procedimiento para generar un DAG. Diferentes modelos prismáticos, los cuales varían en el parámetro de dimensión vertical.

Tal como se revisó en el procedimiento anterior, el funcionamiento de **Dynamo** se basa en un nuevo paradigma denominado *Programación Visual*, el cual es una forma amigable de hacer más eficiente un trabajo que requiere de múltiples modificaciones o que se desea generar una diversidad de propuestas. Sin embargo, aunque se revisó el funcionamiento de este paradigma de programación, este trabajo no planea utilizar en su totalidad este planteamiento. Y esto se debe a que el objetivo que se tiene en mente tiene tal complejidad que el *DAG* resultante constaría de tantos nodos y conexiones que su implementación se vería afectada por el nivel de familiarización del usuario con el software, es decir, que se necesitaría que la persona que utilizaría el programa tenga un nivel avanzado en el uso de **Dynamo** para poder ejecutar los nodos. Y justamente es lo que se desea evitar, ya que mientras más sencillo o amigable pueda ser el software, más útil se puede volver. Por ello se decidió utilizar el módulo de lenguaje *Python* que este software permite utilizar. La forma de programar textualmente se explicará en los capítulos siguientes.

Torre Mayor. Fotografía de la edificación localizada en Ciudad de México que sirvió como principal modelo experimental para este trabajo. A la derecha Torre Reforma. (E. Muttio - 2016)

3.1.2. *Torre Mayor*: Modelado Mediante Parámetros

La geometría de la *Torre Mayor* puede ser tan compleja que se requerirían decenas de parámetros para realizar un modelo realista que considere cada aspecto técnico y estético. Sin embargo, tal como se comentó en la sección anterior, el modelo de la edificación es de tipo conceptual para una fase de planeación temprana. Dicho eso, se buscó la manera de reducir el número de parámetros en lo posible. Para ello se decidió utilizar geometrías compuestas por otras, y los parámetros que el usuario propone asignan las dimensiones de solo una geometría, que en este caso corresponde a la sección rectangular de la edificación, las demás geometrías utilizan estas dimensiones como variables independientes para calcular sus propias medidas utilizando factores.

Se implementaron modelos de edificaciones, entre estos la *Torre Mayor*, generados mediante líneas de código Python, debido a que se deseaba incursionar completamente en la obtención de geometría mediante uso de técnicas de parametrización. Por ello se utilizó las funciones propias de **Dynamo** denominadas *Python Script*, las cuales permiten ingresar parámetros externos a un archivo de texto que será ejecutado por el compilador de Python incorporado. Las figuras 3.13 y 3.14 muestran estos nodos.

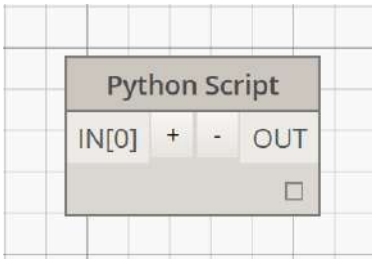


Figura 3.13: Python Script. Función propia de **Dynamo** que permite utilizar un lenguaje de programación textual de alto nivel para automatizar procedimientos dentro del contexto de modelado paramétrico.

La función que se muestra en la figura 3.13 es una implementación en el software Dynamo que tiene un potencial enorme. Ya que además de poder hacer más eficiente un trabajo al realizar una parametrización, como se ha comentado antes, este software permite crear nodos propios utilizando un lenguaje de alto nivel y su librería de geometrías. Para acceder al espacio de programación textual, se debe hacer doble click al módulo y aparecerá una venta con fondo negro, donde ya tiene indicada la importación de la geometría. Esto lo podemos ver en la figura 3.14, sin embargo siempre es recomendable programar utilizando un editor externo y copiar el código dentro de este espacio. Hágase notar que la función *Python Script* tiene botones que permiten agregar parámetros externos que utilizarán las funciones programadas en el interior de estos, su funcionamiento es similar al de *List.Create* revisado antes.

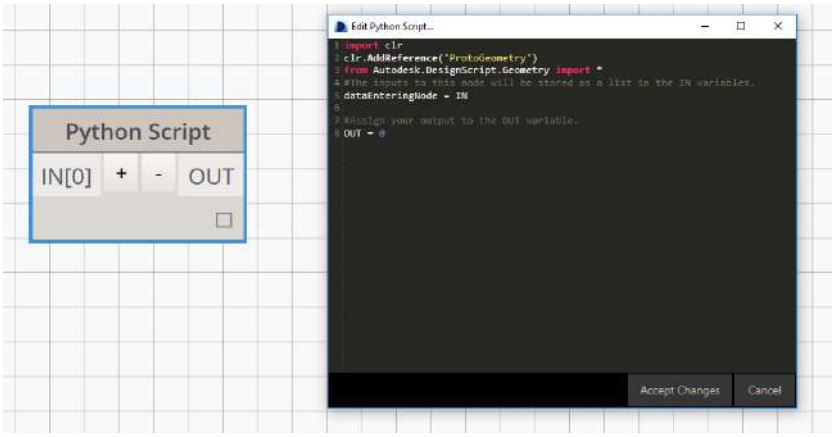


Figura 3.14: Python Script. Acceso al espacio de programación de Python mediante un doble click a la función de **Dynamo**.

A partir de este momento, la mejor recomendación para entender por completo este capítulo será en conjunto con el apéndice B, ya que ahí se encuentran todos los códigos que se implementaron. El código para generar la geometría conceptual de la *Torre Mayor* está en la sección A.1, por lo que para entender la siguiente descripción hay que abrir dicha página donde comienza el código. Entonces como primer paso se ejecuta la función `Torre_Mayor(width, length, height)`, la cual recibe como parámetros tres valores numéricos, ancho, largo y alto. Estos parámetros son los necesarios para generar un prisma rectangular pero serán suficientes para generar toda la geometría de esta famosa edificación. Si separamos las distintas geometrías básicas que componen a la Torre Mayor, encontraremos que la mayoría de sus plantas solamente se tendrá un rectángulo, una semi-elipse y la conexión entre estas dos figuras.

Para lograr generar la totalidad de la geometría de la edificación, se propuso utilizar *factores de proporcionalidad* que se obtuvieron mediante imágenes, fotografías y planos conceptuales. Dichos factores sirven para generar las figuras básicas que componen las plantas, y que sirven como parámetros para las funciones que hacen esta tarea, como lo es generar la semi-elipse y el rectángulo. En cuanto a las primeras plantas las figuras básicas se obtuvieron mediante la figura 3.15, el color rojo muestra el rectángulo mientras el verde tiende a ser una semi-elipse aunque como podemos observar no está completa, sino que es un segmento de ésta. De esta forma se asocian las distancias del segmento rectangular con los radios de la semi-elipse.

Ahora, aparte de las relaciones entre distancias en un plano $X - Y$, podemos observar en la figura 3.16, que la Torre Mayor también tiene variaciones en su geometría de acuerdo a la altura, por ello se obtuvo un factor de distancia vertical entre el edificio rectangular y los demás componentes. Como se puede observar en dicha imagen, los colores marcan las distintas alturas. El color morado denota la altura total del edificio, este parámetro lo proporciona

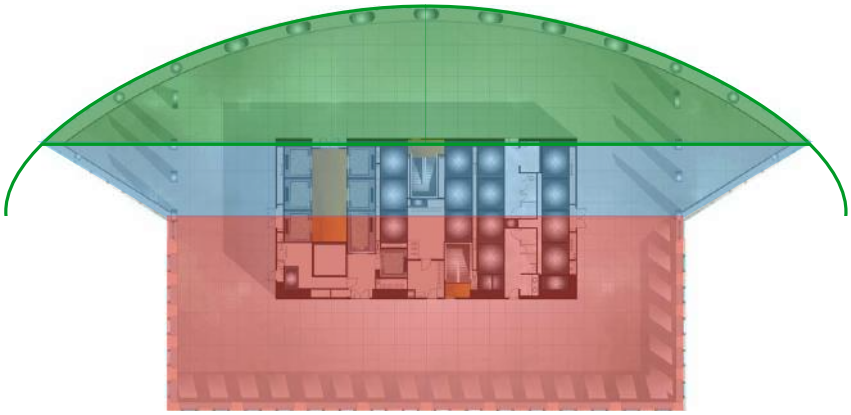


Figura 3.15: Torre Mayor. Planta arquitectónica de los primeros niveles que muestran las figuras básicas que la integran (28).

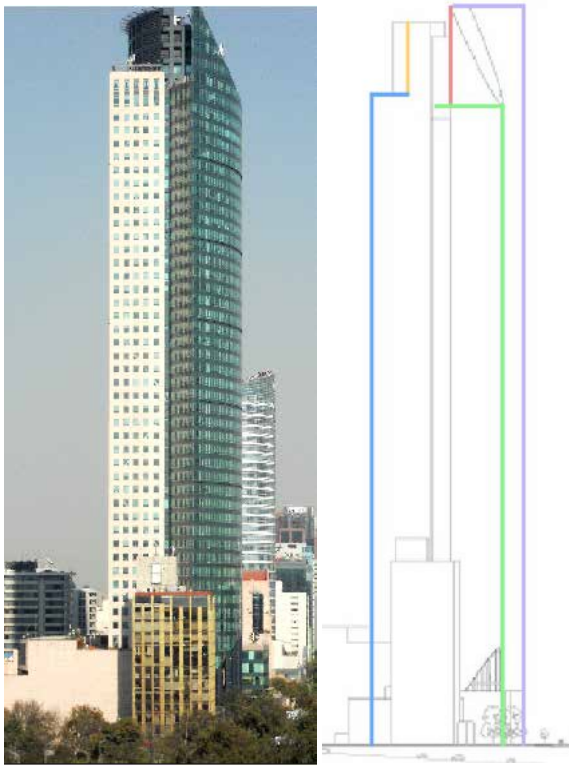


Figura 3.16: Factores de proporcionalidad. Se muestra que hay geometrías distintas con respecto a la altura, la edificación rectangular tiene cierta dimensión vertical mientras que la semi-elíptica tiene otra. Además la figura de la derecha muestra estas distancias con distintos colores (8).

el usuario al hacer la conexión mediante el DAG. Utilizando la magnitud de la altura total, se calcularon los *factores de proporcionalidad* de altura. Como por ejemplo la altura del edificio con forma rectangular que se ve en la imagen 3.16 de color azul corresponde aproximadamente a 0.88078205 de la altura total. En cambio la altura de la edificación semi-elíptica tiene dos segmentos importantes, el primero mostrado con color verde tiene una altura aproximada de 0.98290221 pero con respecto a la altura del edificio rectangular; el segundo segmento con color rojo es la misma semi-elipse pero con un corte o chaflán, el cual tiene una altura de 0.15245245 respecto a la edificación rectangular. Hasta este punto ya se tienen los valores de altura del edificio rectangular, ya que el usuario proporciona las dimensiones en planta y mediante los *factores de proporcionalidad* se obtuvo la altura. Entonces se utilizó una función denominada `rec(width,length, rec_height)` para generar el sólido prismático rectangular. La explicación del código que genera esta geometría es exactamente la misma revisada en la sección 3.1.1, sin embargo en lugar de utilizar una red de conexiones *DAG* con programación visual, se utilizó programación textual con *Python*. Por lo que se mostrará el código y en conjunto con la explicación en dicha sección, se podrá entender fácilmente su funcionamiento.

```
def rec(ancho, largo, altura):

  #Creacion de sistema coordenado en origen
  Identidad=CoordinateSystem.Identity();

  #Creacion de sistema coordenado en punto mas alto

  xc=0;
  yc=0;

  origen=Point.ByCoordinates(xc,yc,altura)
  cs=CoordinateSystem.ByOriginVectors(origen, Identidad.XAxis,
    Identidad.YAxis, Identidad.ZAxis)

  #Creacion de rectangulos (inferior y superior)

  rec_1=Rectangle.ByWidthLength(Identidad, ancho, largo)
  rec_2=Rectangle.ByWidthLength(cs, ancho, largo)

  rec_vec=[]
  rec_vec.append(rec_1)
  rec_vec.append(rec_2)

  #Creacion de solido
  rec_sol=Solid.ByLoft(rec_vec)

  return rec_sol
.
```

Listing 3.1: Función para generar un sólido prismático rectangular.

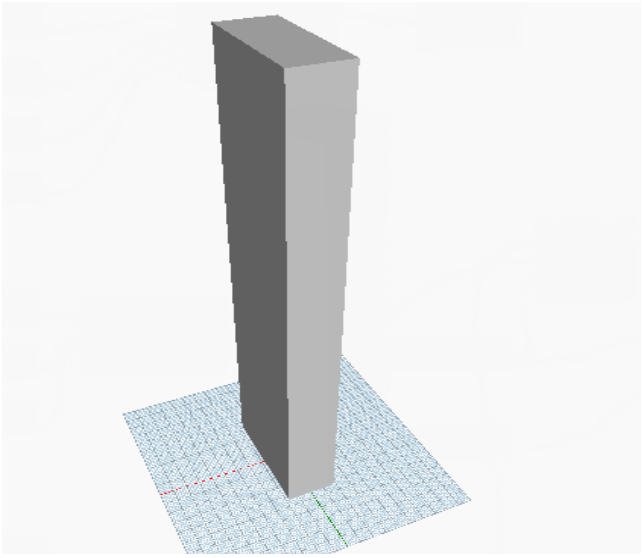


Figura 3.17: Torre Mayor. Componente rectangular de la edificación, se genera el sólido prismático.

El siguiente paso es generar el sólido prismático semi-elíptico, para ello se programó la función `sem_elli_YZ(rad_x,rad_y,elli_height,offset)`. La primer parte de la función genera los sistemas coordenados como se vio antes. La creación de los elipses inferior y superior se pueden hacer mediante una función de **Dynamo** similar a la utilizada en el prisma rectangular, dichas función es `Ellipse.ByCoordinateSystemRadii(cs, rad_x, rad_y)` y devuelve de la misma manera una curva a utilizar para generar el sólido. Lo característico de esta función es que como se comentó antes y tomando como referencia la imagen 3.15, es que la semi-elipse que se planeó en la *Torre Mayor* no está completa, sino que es un segmento de este denotado por un relleno color verde en la imagen. La forma de hacerlo es utilizando un plano de corte perpendicular a la base y a una distancia fija desde del origen. Esta distancia no tiene un parámetro dado por el usuario, sino que se utilizó otro factor de proporcionalidad de acuerdo a la base rectangular. Dicho valor de desplazamiento se denominó *"Offset"*. Entonces se genera el plano $Y - Z$ en el origen y se mueve al punto de corte. Ahí se utiliza la función de **Dynamo**, `Solid.Trim(cut_plane, aux_point)`, la cual necesita del sólido a cortar, un plano y un punto para indicar la parte del sólido que se desea conservar. Entonces mediante un condicional que determina los límites del sólido, se genera el punto auxiliar y para finalizar se hace el corte para tener un sólido con forma de segmento semi-elíptico prismático.

El siguiente componente se muestra en la figura 3.15 de color azul y se puede apreciar tiene la forma de un trapecio. Sin embargo la forma en que se programó es creando dos rectángulos perpendiculares a la base, es decir que los rectángulos están contenidos en el plano $Y - Z$. Para hacer esto se requiere cierta información, y en esta ocasión se implementó usando

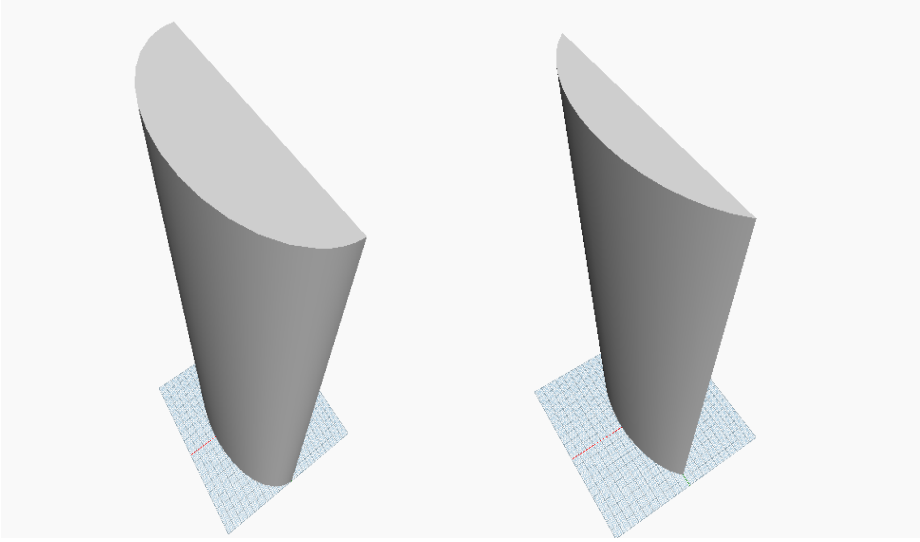


Figura 3.18: Torre Mayor. Sólido prismático semi-elíptico antes y después de que se realiza un proceso de corte mediante un plano definido por el parámetro *Offset*.

la función para creación de rectángulos mediante los puntos de las esquinas `Rectangle.ByCornerPoints(Points[])`. En términos generales se requieren de dos rectángulos, uno que sea adyacente al prisma rectangular de la imagen, y otro que sea adyacente al prisma semi-elíptico verde. Como se puede ver en la imagen, los dos rectángulos tienen dimensiones distintas, por ello se necesitó conocer los puntos extremos de cada figura. Para el primer rectángulo, los puntos necesarios fueron muy sencillo de obtener ya que se utiliza la información proporcionada antes. Entonces para generar por ejemplo el punto inferior derecho, las coordenadas serían $(Ancho/2, Largo/2, 0)$ y el superior $(Ancho/2, Largo/2, Altura)$, de la misma manera para los puntos de la izquierda, pero utilizando las coordenadas $X - Y$ negativas. En cambio, para generar el rectángulo adyacente al sólido del segmento semi-elíptico es un poco más complicado, ya que no se tiene dicha información, sino que se necesita extraer de alguna manera. Lo que se hizo fue lo siguiente, se programó la función que describe una elipse (ecuación 3.1), utilizando la información que conocemos y así es posible calcular las coordenadas de los puntos.

$$y_{elli} = \sqrt{\frac{1 - Offset^2}{Radius_x^2} * (Radius_y^2)} \tag{3.1}$$

Por lo que las coordenadas para el punto derecho inferior son (Offset,y_{elli}, 0) y para el superior (Offset,y_{elli}, $Altura$). De esta manera ya se tienen dos curvas que forman rectángulos y que además son paralelos. Entonces utilizando la función ya descrita antes `Solid.ByLofts(Curves[])`, se genera el sólido con base trapezoidal que une los componentes rectangular y semi-elíptico de

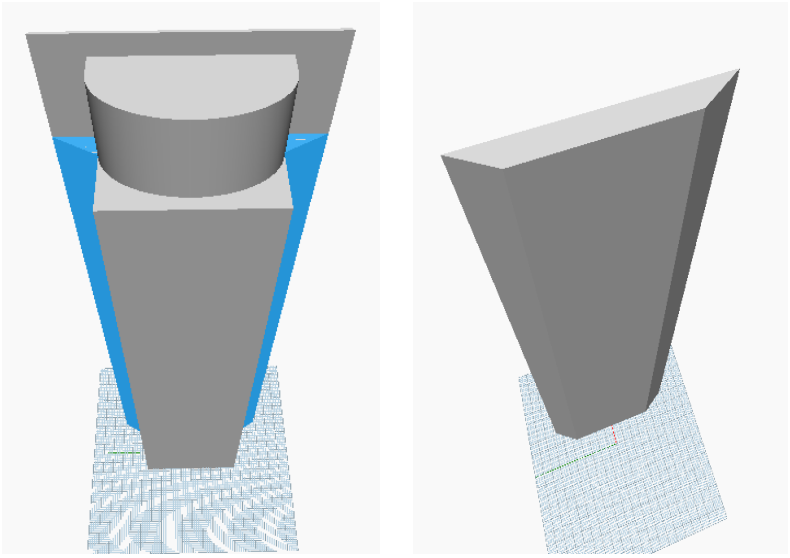


Figura 3.19: Torre Mayor. Unión de los componentes rectangular y semi-elíptico mediante un sólido con base trapezoidal.

la *Torre Mayor*. El próximo sólido a generar se refiere a la continuación del sólido semi-elíptico, pero como podemos ver en la fotografía de la imagen 3.20, el proyecto cuenta con un corte en la parte superior que le da un aspecto estético. El factor de proporcionalidad de este sólido reduce más la altura siendo de 0.1089854 respecto a la altura de la edificación rectangular. La forma de como se implementó la creación de este sólido empieza de la misma manera que la generación del sólido del segmento semi-elíptico, pero antes de retornar la geometría se activa una nueva función que se programó y se denominó `get_chaplane(rad_x,rad_y,height,offset)`. La función genera un plano mediante las coordenadas que se obtienen mediante la fórmula de la elipse mostrada en la ecuación 3.1. La función para crear planos en **Dynamo** es `Plane.ByOriginNormal(Point,Normal_Vector)`, por lo que no solamente se deben obtener las coordenadas en donde se posicionará el plano, sino también un vector perpendicular al plano que se generará. Por lo que los puntos necesarios son los extremos superiores de la edificación y un tercer punto auxiliar que denota el final del plano en la base de la edificación. Posteriormente se utilizó una función que calcula el producto cruz entre dos vectores y por definición el resultado de esta operación es un vector perpendicular a ambos vectores. Entonces ya con esta información se genera el plano de corte y para finalizar se utiliza la función `Solid.Trim(Geometry)` para crear el sólido que se muestra en la figura 3.21.

Ahora para finalizar el modelo conceptual de la edificación *Torre Mayor* se necesita generar un componente sobre el edificio rectangular que en la parte superior alberga un helipuerto. La imagen 3.22 muestra esta parte de la edificación. Para programar la generación de este componente el primer paso es calcular los factores de proporcionalidad respecto a la edificación rectangular

Figura 3.20: Torre Mayor. Fotografía aérea que muestra el corte inclinado del bloque superior de la fachada de la *Torre Mayor* (1).

Figura 3.21: Torre Mayor. Componente superior de la edificación, el cual tiene una forma base con un segmento semielíptico pero además cuenta con un corte tipo "chaflán" que le da un aspecto estético.

Figura 3.22: Torre Mayor. Último componente para la generación de la edificación, éste se muestra con una forma semi-elíptica ya que en su parte superior cuenta con un helipuerto (10).

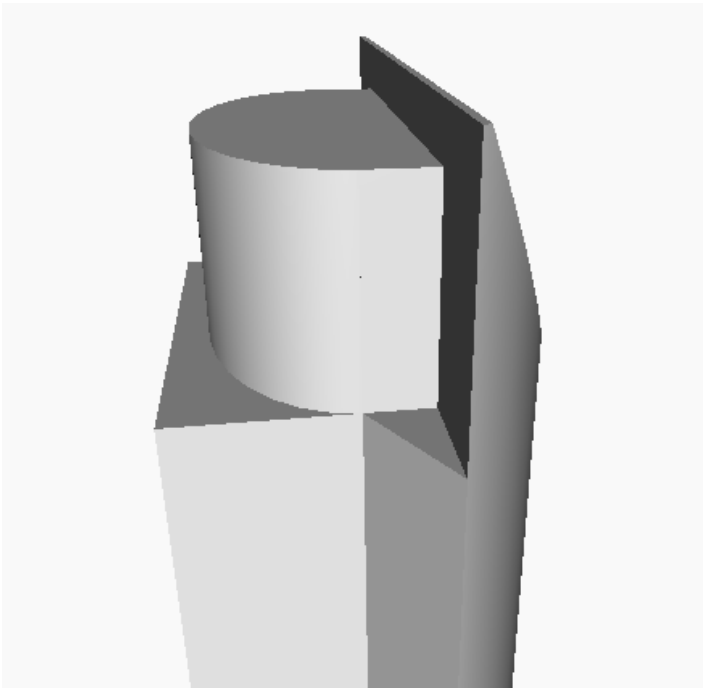


Figura 3.23: Torre Mayor. Componente generado por un sólido semi-elíptico que se encuentra sobre la edificación rectangular.

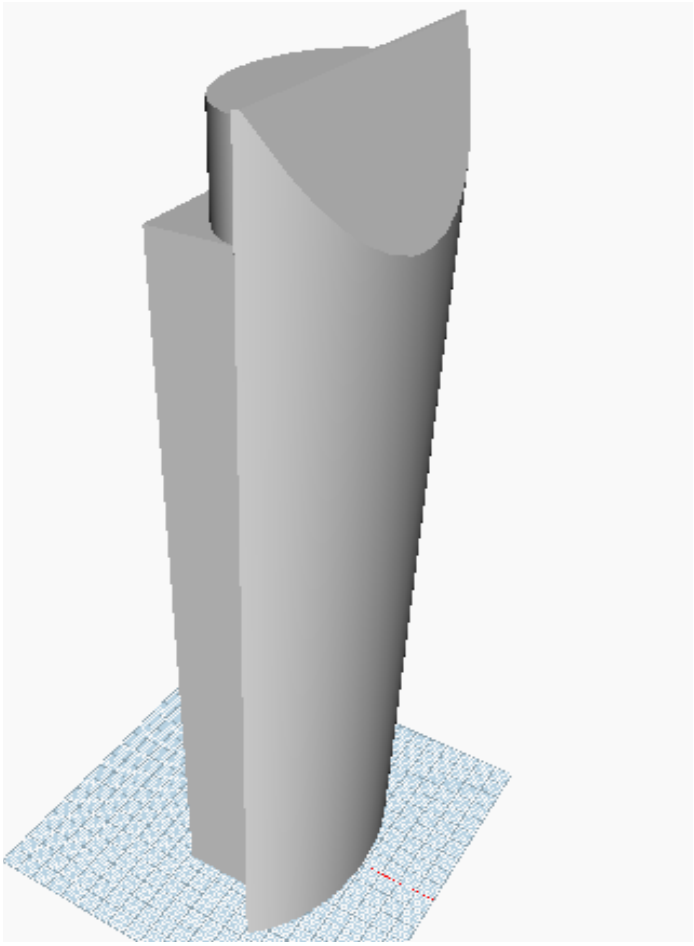


Figura 3.24: Torre Mayor. Modelo conceptual resultante generado utilizando código textual *Python* de la edificación a utilizar como principal experimento en este trabajo.

como en los otros componentes. Posteriormente se utiliza la misma función para la obtención del sólido semi-elíptico prismático utilizado y explicado en los componentes anteriores. La única diferencia es una operación de desplazamiento propia de **Dynamo** denominada `Geometry.Translate(x,y,z)`, para colocar en su posición final esta parte de la edificación. El modelo tridimensional se muestra en la figura 3.23. Al tener todos los componentes generados y en posición, el último paso será unirlos en uno solo, ya que por ahora solo se tienen modelos separados, los cuales no servirán para los procesos de estructuración que se tienen planeados. La función utilizada para unir los sólidos es "`Solid.UnionAll(Solids[])`". El modelo conceptual completo de la edificación *Torre Mayor* se muestra en la figura 3.24. Un aspecto importante de mostrar la creación de este modelo es que es éste el sólido que será el experimento principal para los módulos de estructuración que se desarrollaran durante este capítulo, la prueba positiva de este modelo dará pie a extrapolar los beneficios en otras edificaciones debido a los componentes que posee.

3.2. Intersección de Sólido Tridimensional Mediante Planos

Él código del módulo ***Floors*** se programó con una estructura similar a la que presentaría un programa realizado en lenguaje de bajo o mediano nivel como por ejemplo *Lenguaje C*, ya que se implementaron funciones y se asignó una sección del script a la ejecución de dichas funciones, esta sección se denominó *Main* tal como se conoce a la función principal de un lenguaje tipo *C o C++*.

La primer función del código que se implementó se llama `get_floor_curves (solid, height, n_floors)`. Su función principal es generar una lista de objetos *superficies*, los cuales son proporcionados por la biblioteca geométrica de Dynamo, éstos formarán los niveles idealizados de la edificación como planos. Los argumentos necesarios son el sólido tridimensional modelado por el usuario, del cual se habla en la sección 3.1; la altura de la edificación indicada con un número flotante, el cual también puede ser introducido mediante un *"Number Slider"* y el cual debe corresponder con la altura del modelo tridimensional; y por último el número de pisos deseado.

En general, el procedimiento es designar un vector con las alturas en distancia vertical Z de cada nivel a generar, con este se utiliza la función propia de Dynamo `Plane.ByOriginalNormal(Point, Vector)` para obtener un vector de planos mediante el centroide de cada superficie y un vector con dirección a $+Z$. Posteriormente el algoritmo es muy básico ya que solo se necesita hacer una intersección del sólido con el plano correspondiente a cada nivel, de hecho solo es necesario un ciclo que recorra el vector de planos y una línea de código correspondiente a la intersección mediante la función de Dynamo `Solid.Intersect(Geometry)`. El siguiente extracto del código describe el procedimiento mencionado antes. Utilizando un sólido de tipo prisma rectangular extraído de los componentes básicos que integran la edificación "Torre Mayor", se muestra el resultado de la intersección que se muestra en el código para obtener los niveles en la imagen 3.25.

```
1
2   for i in floor_planes:
3
4     floor_inter.append(PolySurface.ByJoinedSurfaces(solid.
      Intersect(i)))
5 .
```

Listing 3.2: Obtención de superficies mediante intersección de sólido.

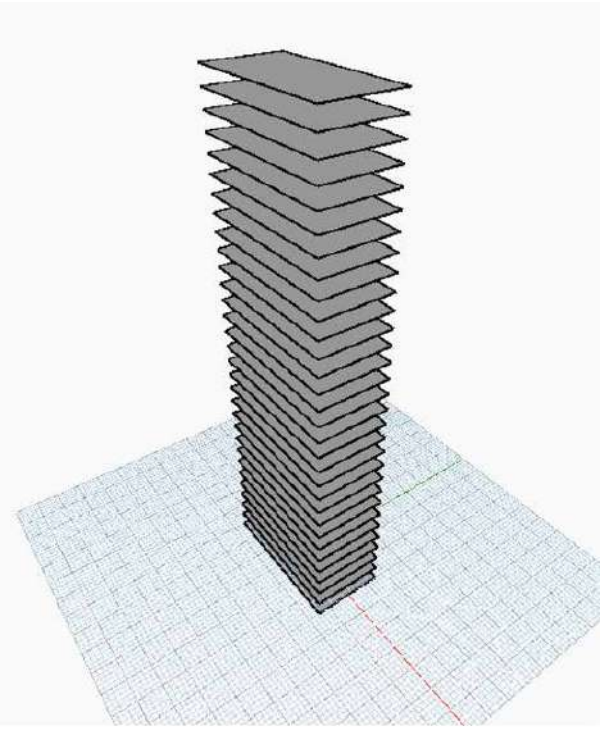


Figura 3.25: Intersección de sólido. Obtención de los niveles de la edificación mediante una operación de intersección de planos a un sólido tridimensional.

Sin embargo en este punto se presenta un costo computacional muy alto debido a que la operación de intersección de geometrías de alto orden como lo son las superficies y los sólidos necesitan de poder de cálculo alto. Este proyecto se ha planeado para que pueda ser utilizado en computadoras de fácil acceso, tal como se menciona en el capítulo 2 se debe hacer un balance entre la eficiencia de acuerdo al tiempo y complejidad de código. Se hicieron pruebas de tiempo o latencia en el modelo experimental *Torre Mayor*, y se tomaron en cuenta tanto la intersección de superficies en el sólido para la generación de niveles que realiza el módulo ***Floors*** explicado en esta sección, así como características de estructuración que se explicarán en la sección 3.3. Aún cuando no se ha revisado la forma de estructurar, se decidió mostrar este resultado debido a que éste es el primer módulo al que se determinó hacer una optimización. Por ello, la gráfica mostrada en la figura 3.26 denota un crecimiento gradual del número de intersecciones desde 5 a 30 correspondiente al mismo modelo tridimensional (*Torre Mayor*). Además también se le incrementó el número de barras. Como podemos ver, la gráfica tiene un comportamiento cuasi-exponencial, mientras más elementos son solicitados por el usuario, el tiempo aumenta de manera muy grande. Entre los primeros niveles $0 - 10$ casi no existe una diferencia, pero en 30 niveles la latencia es de aproximadamente 1600 segundos. Esto es bastante, por lo que la solución propuesta fue hacer una optimización mediante técnicas de paralelización.

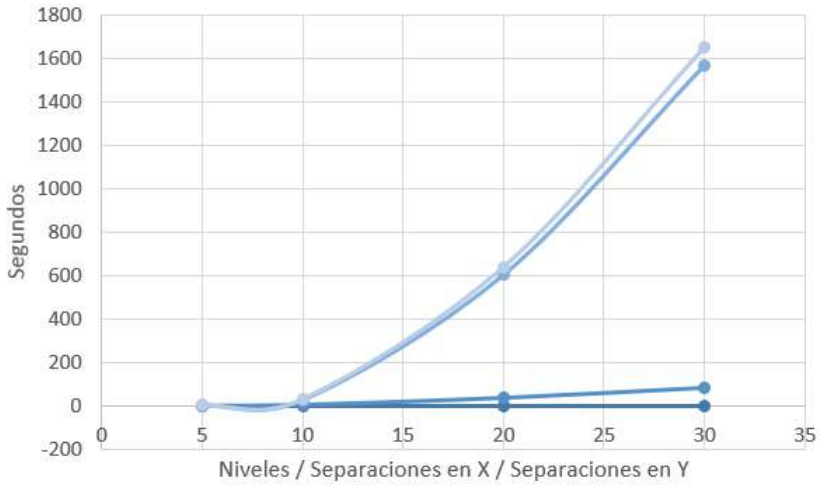


Figura 3.26: Latencia de Intersección. Se muestra una gráfica de tiempo contra complejidad del modelo experimental *Torre Mayor*. Se incrementó gradualmente el número de niveles y número de separaciones del ***Grid*** en ambas direcciones hasta llegar a un valor máximo igual a 30. Las distintas líneas mostradas corresponden a distintas funciones que se involucran en el proceso.

3.2.1. Programación en Paralelo

Tradicionalmente, los programas de computadora se escriben de una manera lineal, es decir, un problema se divide en series discretas de instrucciones para que posteriormente sean ejecutadas secuencialmente una tras otra por un solo procesador, solo una instrucción podía ser ejecutado por cada momento de tiempo. En comparación la *Programación en Paralelo* es el uso de múltiples procesadores para resolver un problema, dividiendo el número de tareas por cada recurso existente, es decir:

- Un problema se divide en partes que puede ser resuelto simultáneamente.
- Cada parte es dividida en una serie de instrucciones.
- Las instrucciones se ejecutan simultáneamente en distintos procesadores.
- Un mecanismo de coordinación y control se hace necesario.

¿Porqué se usa la Programación en Paralelo?

El mundo real es masivamente paralelo, en la naturaleza existen muchos eventos complejos, interrelacionados y que están ocurriendo al mismo tiempo. A comparación con una computación serial, este paradigma de paralelización permite realizar mejores modelados, simulaciones y permite el entendimiento de fenómenos complejos, como por ejemplo la formación de nuevas galaxias, los movimientos planetarios, el cambio climático, el tránsito vehicular, los movimientos de tierra, entre otros.

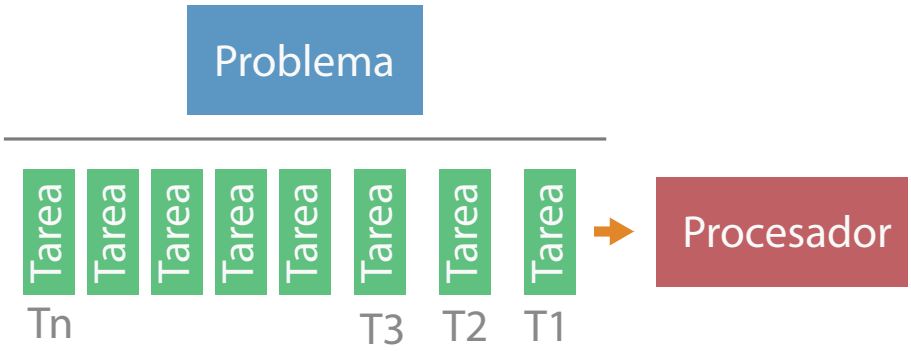


Figura 3.27: Programa Lineal. Programación de manera tradicional, utilizando el paradigma estructurado lineal, las tareas se ejecutan por un solo procesador. (Figura adaptada de (5))

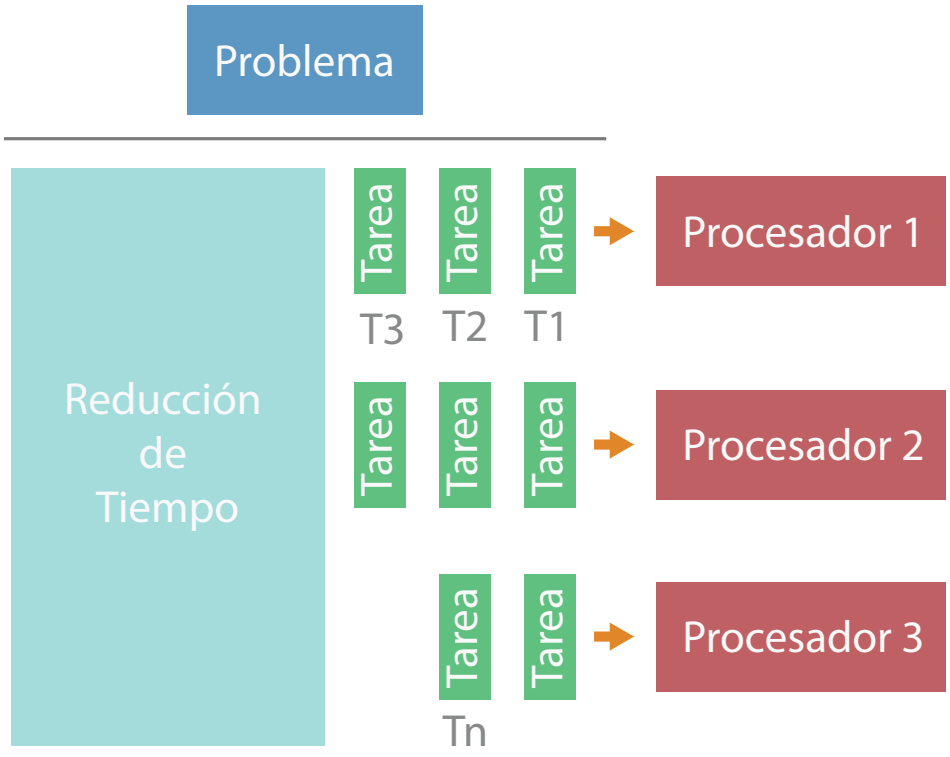


Figura 3.28: Programa Paralelizado. Resolución de tareas de forma simultánea mediante la implementación de la programación en paralelo con distintos procesadores. (Figura adaptada de (5)

Una de las principales razones es por que se puede ahorrar tiempo y dinero. En teoría, al reducir el tiempo de ejecución, se está ahorrando en costos. Además las computadoras con estos sistemas están al alcance de todos. Las computadoras, incluso laptops, tienen una arquitectura con múltiples "cores". Se podría decir que la programación lineal en las nuevas computadoras es un "gasto" de potencial en hardware existente. Por otro lado, existen otras razones como por ejemplo la solución de sistemas muy grandes y complejos que son imposibles de resolver con una sola computadora y limitada memoria, tan solo los motores de búsqueda en línea procesan millones de transacciones en un segundo (5). Sin embargo el uso de este paradigma involucra algunos pasos que se omiten en una programación lineal. Esto es debido a que los dominios del problema son divididos y existe el problema de que hay algunas fases de la resolución en la que éstos necesitan compartir información o incluso volver a reunirse. Es por ello que se han desarrollado técnicas que benefician la lógica de implementación de algoritmos paralelizados, ya que estas etapas son costosas debido a la memoria que deben compartir.

¿Quiénes necesitan usar la Programación en Paralelo?

Este paradigma de programación es altamente empleado en *Ciencia e Ingeniería*, ya que es necesario en problemas difíciles de resolver como:

- Atmósfera, Ambiente, Geología y Sismología..
- Física aplicada, nuclear, de partículas, de fusión y fotones.
- Bio-Ingeniería, Bio-Tecnología y Genética.
- Ingeniería Mecánica, desde prótesis hasta naves espaciales.
- Entre muchas otras.

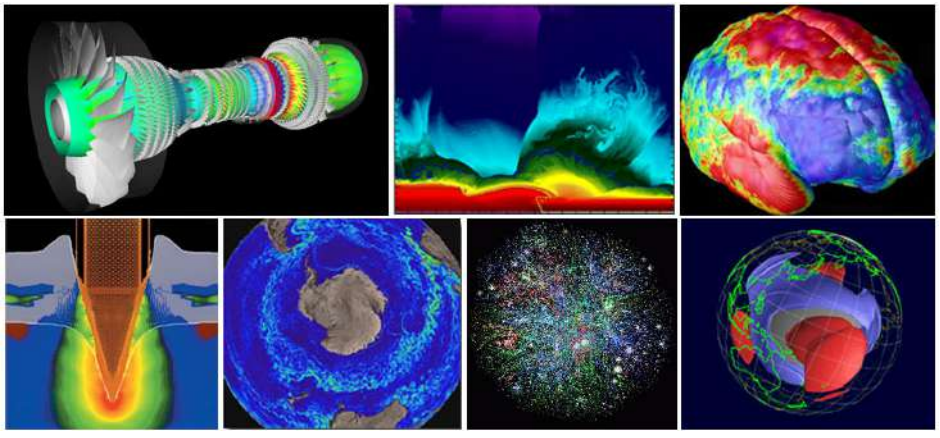


Figura 3.29: Aplicaciones de la Programación en Paralelo. Ejemplos de aplicación en Ciencia e Ingeniería (5).

La forma de realizar una programación en paralelo en la intersección de superficies para los niveles de las edificaciones, fue utilizando una librería implementada en *Python* con el nombre *Threading*, la cual está especializada para hacer una ejecución de código en paralelo de forma muy sencilla, se cambio la estructura inicial del código asignando superficies de los niveles a diferentes hilos que harán simultáneamente el cálculo geométrico. La figura 3.30 muestra de forma gráfica como se eligió realizar la división de plantas, se observa que ésta depende del número de niveles que se eligieron por el usuario, para ilustrarlo se utilizó el mismo sólido prismático de la imagen 3.25. Entonces, el número de intersecciones de superficies con el sólido tridimensional (que recordemos es una operación geométrica fuerte) que realiza un solo núcleo de la computadora, es reducido considerablemente.

Con esta nueva implementación se hicieron pruebas de tiempo o latencia, las cuales dieron un impacto positivo al considerara un número muy grande de niveles. Sin embargo, ocurrió un comportamiento peculiar al considerar pocos pisos. Cuando se solicitaba un número de niveles menor a 10 se presentaba un *incremento de tiempo* respecto a la estructura del código *no paralelizada*. Esto, aunque no se esperó inicialmente, tiene mucha lógica, lo podemos observar comparando el primer código, donde observamos que por cada iteración se hace una intersección y nada más, es decir *procedimiento de una etapa*. Ahora el concepto de paralelización, tal como se describe en el código que se presenta a continuación, requiere de más etapas en el procedimiento:

1. Separación de regiones de cálculo.
2. Asignación de cálculo que realizará cada núcleo.
3. Integración de las regiones separadas anteriormente.

Entonces, el nuevo código tiene que separar niveles, asignarles la operación que en este caso es una intersección de superficie a un sólido tridimensional, y al final se deben acomodar nuevamente las partes separadas. La cuestión es que no se puede simplemente guardar estas regiones en un espacio de memoria y ya. Sino que se deben ajustar de acuerdo a la estructura que se tenía inicialmente, por ejemplo, en este caso los niveles necesitan estar acomodados de forma ascendente, mediante su coordenada en Z vertical, ya que los procedimientos que se harán posteriormente se podrían volver más complicados al tener un orden aleatorio en los niveles. Pero *¿porqué tendríamos un orden aleatorio en los niveles, si estos se obtienen en un orden ascendente?* Bueno, pues resulta que de acuerdo a los conceptos de paralelización, vistos antes, tenemos que el procesador de la computadora asigna el ordenamiento de las tareas según disponga de recursos, por lo que al tener cálculos realizándose simultáneamente y separados por núcleos, algunos "tendrán prioridad de cálculo" y la mayoría de estas ocasiones pareciera que no tuvieran un orden lógico. Es por ello que

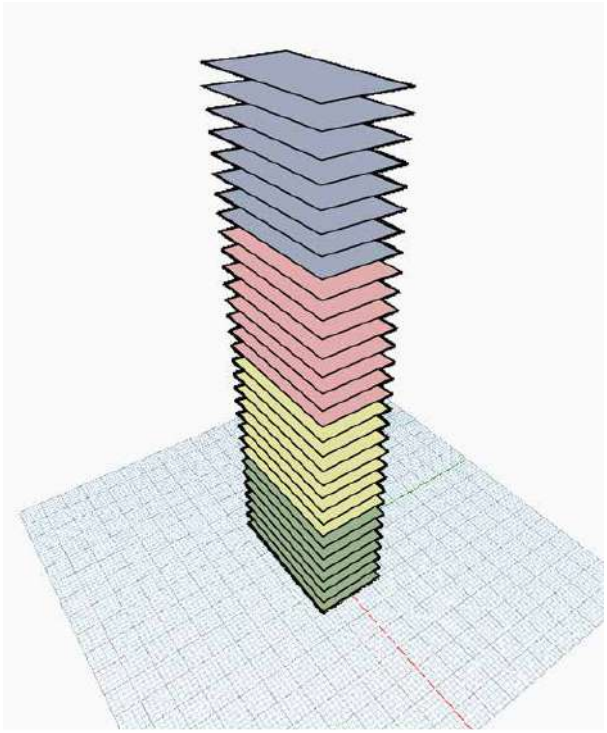


Figura 3.30: Paralelización de intersecciones. Utilizando el mismo concepto mostrado en la figura 3.25, se ilustra la implementación a cuatro núcleos de la programación en paralelo.

al final de hacer el cálculo de las intersecciones, se implemento un algoritmo de ordenación que de acuerdo a la coordenada Z, acomoda los niveles recién creados. A continuación se presenta el algoritmo que incluye la paralelización en la intersección de superficies.

Podemos observar que en la etapa tres correspondiente a la integración de los niveles, se utilizó una función nativa de Python denominada `sorted()`, la cual hace el trabajo de ordenación. Al principio se implementó un algoritmo basado en *ordenación "burbuja"*, pero al contrario de lo que se pensaba, no fue tan eficiente como *sorted*, ya que erróneamente se creía que por ser una función proveniente de un lenguaje de alto nivel su ejecución sería lenta. Al darle una oportunidad, se corroboró su excelente desempeño y además su fácil implementación al código, esta es una de las muy buenas características que Python tiene para su aprovechamiento.

```
1
2  #Libreria necesaria para realizar paralelizacion en Python
3  from System.Threading import Thread, ThreadStart
4
5  #len1 es el numero de niveles que se repartira a los cuatro arreglos (
     por cada nucleo)
6  len1 = int(round(len(floor_planes)-1 ) ) / 4 + 1
7
8  #Un arreglo compuesto por cuatro sub_arreglos de tamanio len1
9  floor_planes_chop=[floor_planes[x:x+len1] for x in xrange(0, len(
     floor_planes)-1, len1)]
10
11 #Arreglo de cuatro sub_arreglos vacios que contendra las superficies
     generadas
12 floor_inter_aux=[None,None,None,None]
13
14 #Cuatro funciones debido a los cuatro nucleos a utilizar
15 def threaded_ps0():
16   for i in floor_planes_chop[0]:
17     floor_inter_aux.append(solid.Intersect(i))
18   return floor_inter_aux
19
20 def threaded_ps1():
21   for i in floor_planes_chop[1]:
22     floor_inter_aux.append(solid.Intersect(i))
23   return floor_inter_aux
24
25 def threaded_ps2():
26   for i in floor_planes_chop[2]:
27     floor_inter_aux.append(solid.Intersect(i))
28   return floor_inter_aux
29
30 def threaded_ps3():
31   for i in floor_planes_chop[3]:
32     floor_inter_aux.append(solid.Intersect(i))
33   return floor_inter_aux
34
35
36 ## Algoritmo de paralelizacion en mas de 10 pisos
37 #Se encontro que para menos de 10 pisos, no es eficiente utilizar
     programacion en paralelo.
38 if n_floors>10:
39
40   #Esto es debido a que algunas veces por el numero de niveles, solo
     se necesitan tres arreglos
41   if floor_planes_chop.Count == 3:
42
43     #La funcion Thread ejecuta simultaneamente las funciones por cada
     nucleo
44     threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(
     threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ))
45
46   else:
47     threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(
     threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ),Thread(
     ThreadStart(threaded_ps3) ) )
48
49   #Declaraciones necesarias para ejecutar las funciones
     simultaneamente
50   for t in threads: t.Start()
51   for t in threads: t.Join()
52
```

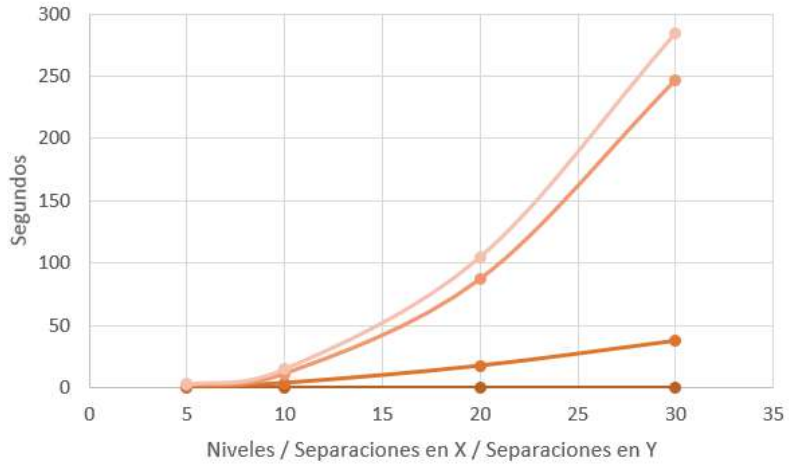


Figura 3.31: Latencia de Intersección. Se muestra la gráfica de tiempo contra complejidad del modelo experimental *Torre Mayor* vista antes en la figura 3.26. Se observa una latencia menor utilizando el concepto de programación en paralelo, por lo que es recomendable usarlo para optimizar el código.

```
53      #Verificacion de valores nulos recibidos y que se desecharan
54      for i in range(floor_inter_aux.Count):
55        if floor_inter_aux[i] is not None:
56            floor_inter.append(PolySurface.ByJoinedSurfaces(
        floor_inter_aux[i]))
57
58      #Organizacion de niveles desde el inferior hasta el superior
59      for i in range(floor_inter.Count):
60        p_aux=floor_inter[i].PointAtParameter(0.0,0.0)
61        aux_point_parameter.append([floor_inter[i],p_aux.Z])
62
63      floor_inter_pre = sorted(aux_point_parameter, key=lambda tup: tup
        [1])
64      floor_inter=[]
65      for i in floor_inter_pre:
66        floor_inter.append(i[0])
67
68    else:
69      #Si son menos de diez niveles es recomendable utilizar la funcion
        sin paralelizacion
70      for i in floor_planes:
71
72        floor_inter.append(PolySurface.ByJoinedSurfaces(solid.Intersect(i)
        ))
73  .
```

Listing 3.3: Algoritmo de paralelización utilizado para la intersección del sólido.

Posteriormente de realizar el cambio de algoritmo lineal de una sola línea visto antes, al algoritmo paralelizado se hizo la misma prueba de latencia con respecto al incremento del número de intersecciones y niveles, la gráfica resultante (mostrada en la figura 3.31) tiene un aspecto muy positivo ya que se redujo considerablemente el tiempo de ejecución.

3.2.2. Curvas Perimetrales

Ahora, además de obtener las superficies de los niveles es necesario sustraer más información a partir de éstas para generar la estructura exterior. Es decir, necesitamos conocer los límites de cada nivel para saber donde se colocaran vigas que lo soportaran. Para ello se programó una función a la que se denominó `get_floor_per_curves(poly_surf)`, la cual tiene como único parámetro un vector de superficies organizadas de acuerdo a la coordenada Z vertical, y su tarea es obtener las curvas perimetrales que describen la frontera de cada nivel. Para ello se utilizó una función proveniente de la librería de **Dynamo**, llamada `Surface.PerimeterCurves()` la cual regresa una curva, sin embargo se utilizó esta función dentro de un ciclo que recorre cada nivel por lo que la implementación exacta puede verse en el código completo.

Ya que las curvas generadas por cada nivel pueden no ser continuas, es decir, que un nivel tenga un conjunto de curvas en lugar de una sola curva perimetral, se programó otra función para que las fusionara y las guardara en un elemento geométrico denominado *Polycurve o Policurva*. Este caso ocurre sobretodo al utilizar sólidos compuestos, donde dichas intersecciones de figuras crean vértices que la función `Surface.PerimeterCurves()` separa en partes. Debido a que se desea tener el mismo comportamiento para plantas de edificaciones con la misma geometría y además incluir sólidos compuestos, se programó la función `get_floor_pol_curves(peri_curves)` la cual simplemente fusiona las curvas perimetrales de un mismo nivel en una sola, para ello se hace un manejo de listas ya que cuando existen más curvas en un nivel se crean sublistas que deben descomponerse mediante ciclos anidados. El procedimiento se muestra en la imagen 3.32 y se puede describir de la siguiente manera:

1era. Etapa Utilización de geometrías base con una dirección de líneas preestablecida por **Dynamo**.

2da. Etapa Intersección de sólido mediante planos perpendiculares para obtención de niveles.

3era. Etapa Obtención de líneas perimetrales (separadas), que continúan con la numeración de origen.

4ta. Etapa Se agrupan y unen las diferentes líneas en una "Poli-línea" utilizando un arreglo.

El *output* de este primer módulo denota la forma en que se entregará información por cada *DAG* de ahora en adelante. El primer elemento que se retorna del módulo ***Floors*** es una lista denominada *conec_parameters* debido a que aquí se ingresara la información proveniente de un *input* anterior (que en este caso solamente se refiere al parámetro de altura de edificación y al número de

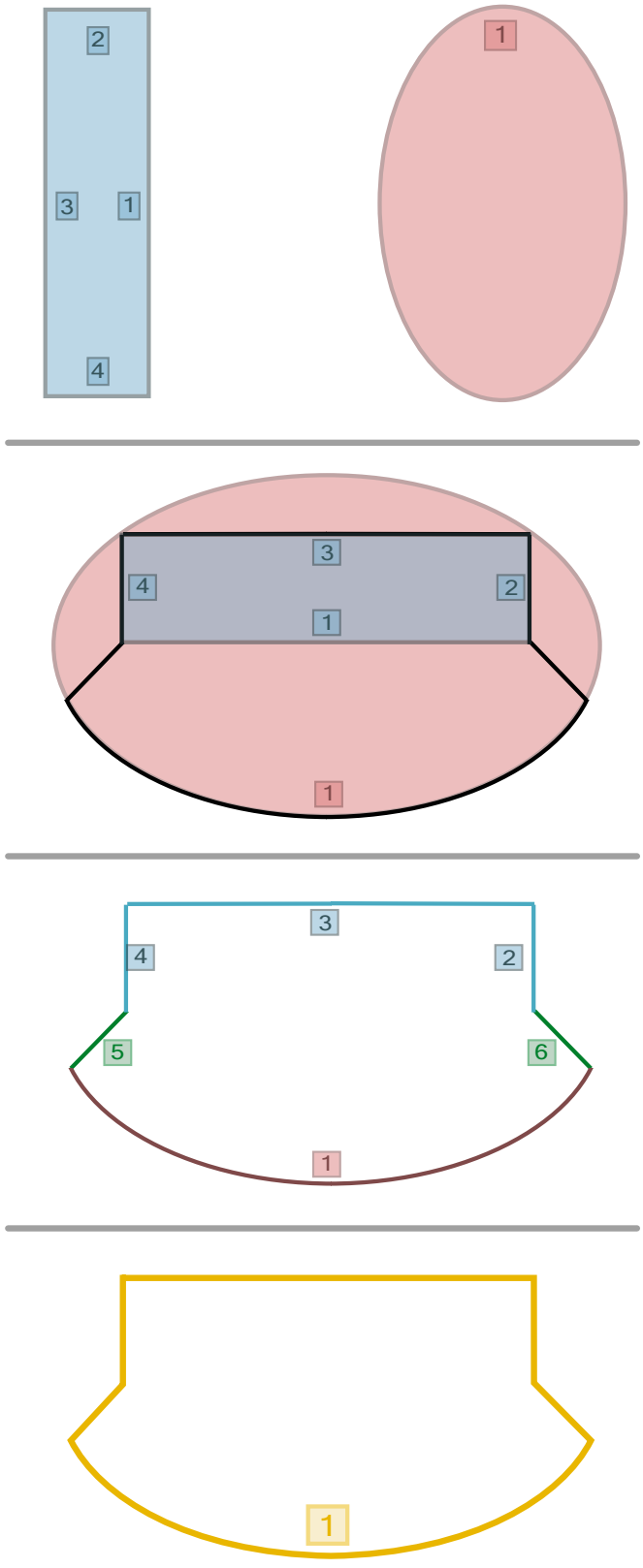


Figura 3.32: Curvas Perimetrales. Procedimiento para generar las líneas perimetrales de cada planta. El ejemplo mostrado corresponde a la planta de la *Torre Mayor*, y se puede observar que la dirección y numeración de las líneas de geometrías base se conserva incluso cuando se genera un sólido compuesto. Esto dificulta procesos posteriores de estructuración.

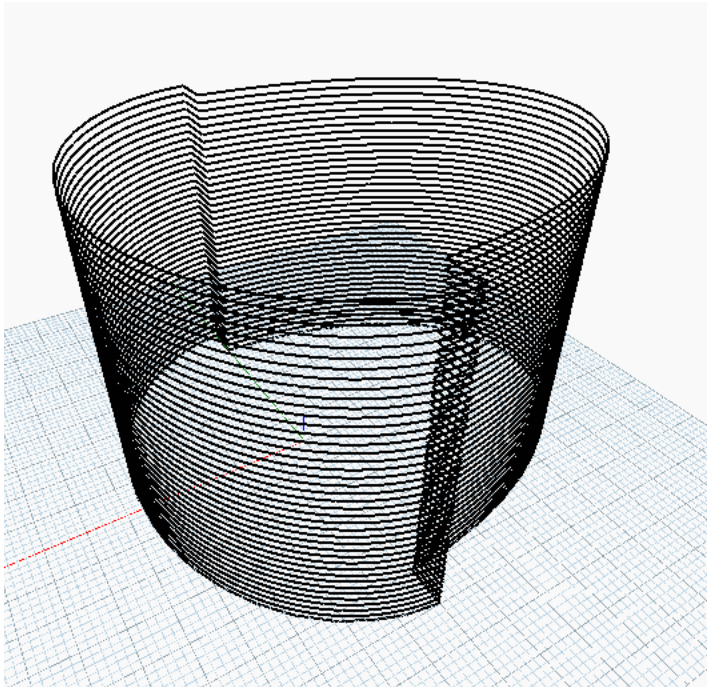


Figura 3.33: Curvas Perimetrales. Imagen que muestra como se generan las líneas perimetrales por cada nivel de una edificación.

niveles considerados). Se decidió tener una estructura de esta manera, ya que de otra forma los módulos posteriores que necesitaran información de algún parámetro inicial tendrían que solicitarla mediante una conexión en el propio *DAG*, esto implicaría tener al final una red de conexiones que incrementan el valor de complejidad ciclomática y por ende es menos amigable la utilización de la aplicación. Por ello se decidió tener conexiones en su gran mayoría paralelas. Un módulo 1 recibe información y exporta un arreglo que conecta al módulo 2 siguiente, sin tener que cruzar conexiones. El segundo arreglo a retornar es el que contiene las superficies de los niveles y por último el tercer elemento es una lista de policurvas que describen el perímetro de cada nivel.

3.3. Configuración Estructural Basada en una Retícula Base

Tal como se ha mencionado antes, la propuesta de configuración de una edificación depende de muchos factores y parámetros que solamente una persona suficientemente capacitada para realizarlo puede estar seguro de su funcionalidad y aún más importante de la seguridad que brindará. La idea de realizar esta aplicación es reducir los costos y tiempos debido a los cambios, lo más cercano a cero que sea posible, además de poder revisar múltiples configuraciones en menor tiempo. Para lograr esta meta, es claro que se necesita de definir algunas reglas, reducir factores y simplificar el modelo disminuyendo el número

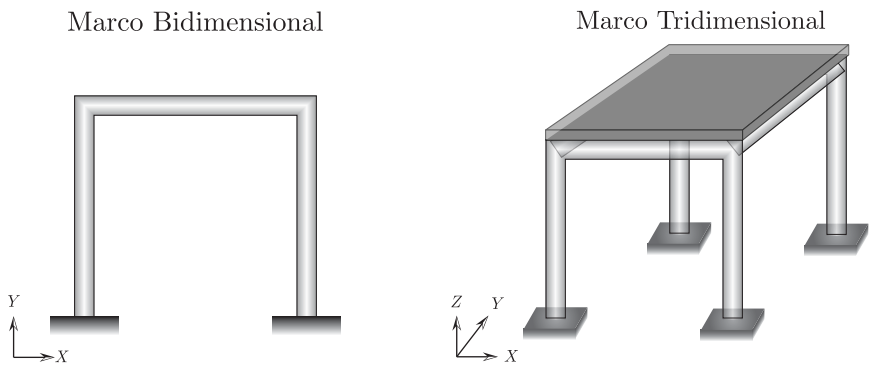


Figura 3.34: Estructuración. Pórtico o marco en un plano bidimensional y uno tridimensional para denotar el tipo de estructuración empleada en este trabajo.

de parámetros a tal grado que permita su optimización en un menor campo de búsqueda. La metodología utilizada también se pensó que fuera escalable, en otras palabras, de fácil modificación para mejorarla, y considerar otros casos particulares. Por esta razón se hicieron módulos separados, de esta manera si alguien tiene una inquietud de hacer más grande el número de modelos a considerar, puede emplear algunos módulos descritos aquí y los que no se necesiten se puede prescindir de ellos.

Dentro del contexto utilizado, se pretende analizar y verificar el diseño de edificaciones utilizando elementos marco tridimensionales y suponiendo uniones rígidas entre las barras. Entonces de acuerdo a la metodología de análisis utilizada, se requiere definir nodos, los cuales conectan barras y a su vez éstas forman pórticos que al multiplicarse se obtendrán niveles y el conjunto de todos estos elementos describirán la edificación a analizar.

El módulo denominado ***Grid*** es quizá en donde el diseñador tendrá la mayor interacción de todo el procedimiento planteado. Se debe a que en este punto se definirá la posición de los elementos estructurales principales. Una pregunta indispensable consistía en saber que tipo de configuración podría proponer un usuario, debido a que dependiendo el análisis a realizar o factores constructivos como el tipo de losa o elementos de apoyo puede ser diferente. Por lo que se decidió que los marcos rígidos sería la mejor opción debido a que son altamente empleados en estructuras de grandes dimensiones. Los marcos o pórticos se componen simplemente de dos columnas que sostienen a una viga al encontrarse representados en un plano bidimensional, y al considerar uno tridimensional simplemente se debe replicar para formar un *cubo o un prisma rectangular* como se muestra en la figura 3.34. Entonces, se debía pensar la forma más sencilla de incorporar este planteamiento a una entrada de datos mediante parámetros.

El planteamiento más convincente que se propuso debido a los pocos parámetros que se necesitan, es utilizar una retícula de ayuda en donde se puedan visualizar puntos, que es donde estarán localizadas geométricamente las columnas, ya que estos elementos estructurales definen la conectividad de las vigas que soportarán la losa y por consiguiente conseguir la geometría de la configuración estructural. En la imagen 3.35 se muestra el planteamiento requerido en base a una retícula de ayuda, la que se denominó ***Grid*** y denota la posición de las columnas que integran los marcos, además se muestra posibles variaciones de la posición de dichas columnas. Podemos observar que para una edificación con una planta rectangular o cuadrada, este planteamiento es muy sencillo de considerar. Los parámetros necesarios para obtener una retícula formada por puntos simplemente es el tamaño de ésta, la cual sería definida como un rectángulo, es decir, tamaño en dirección X y tamaño en dirección Y (de acuerdo al sistema que muestra la figura), y además los renglones y columnas de puntos que podemos definir mediante el número de "nichos" o "pórticos" que queramos considerar.

Con este planteamiento, tenemos que si utilizamos una retícula con tamaño igual a la planta rectangular de nuestro edificio los parámetros que permitirán que exista diversidad en el diseño serán únicamente el número de pórticos en ambas direcciones. Esto nos daría como resultado vigas en un sentido de la misma longitud y en la otra dirección tendríamos quizá otro valor de longitud pero que replicaría a lo largo de este sentido. Para ello se decidió darle un poco más de flexibilidad al considerar que la retícula puede tener dimensiones más grandes que la planta. Las implicaciones de hacer esto es que algunos puntos ayuda que indican la posición de las columnas, ahora se encuentran fuera de la planta, figura 3.36. Y con ello se podrían generar vigas que salgan de la planta de la edificación. La mejor idea fue restringir a que toda planta tendrá una estructuración exterior, y para que funcionara se tuvo que implementar una intersección de las vigas que salen de ésta con las líneas que definen el perímetro. De esta manera se obtiene un punto como resultado de la intersección anterior y por ende este punto servirá para generar las columnas exteriores. Debido a este planteamiento, el módulo tiene la limitación de no poder proponer vigas en "cantilever" o "voladizo", a menos que la planta superior tenga un excedente de área con respecto a la inferior. Sin embargo, el punto positivo es que ya podemos darle una variabilidad más a la designación de la estructura y con ello obtenemos más flexibilidad de diseño.

Al igual que en el módulo ***Floors***, el módulo denominado ***Grid*** funciona mediante una organización similar a un código de lenguaje de más bajo nivel. Por lo que tenemos un *Main* que inicia creando algunas *variables* nuevas y asignando algunas a partir de recibir la conexión de parámetros del módulo anterior, ver apéndice A.3. Ya que aquí también se ocupan las variables como por ejemplo "número de niveles" o "Altura", éstas se re-definen utilizando la

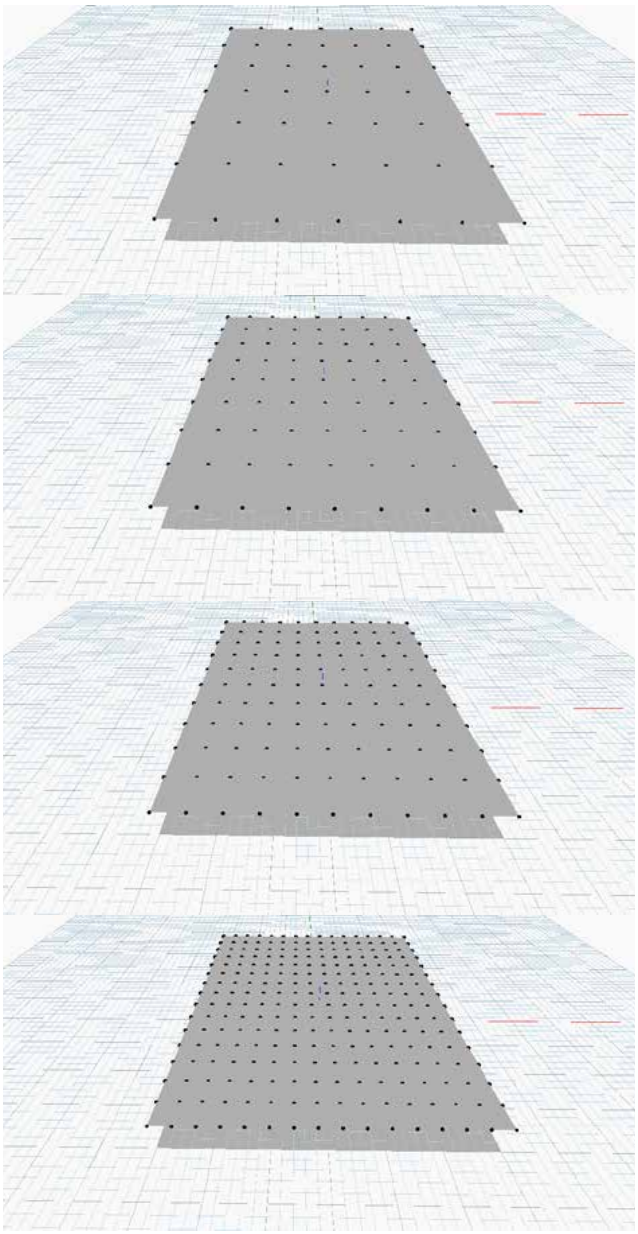


Figura 3.35: Grid. Estructuración de una edificación mediante una retícula o *"Grid"* que muestra la posición de las columnas que formarán los marcos estructurales. Además se muestra la misma planta rectangular, con un grid del mismo tamaño que ésta pero variando el número de separaciones entre columnas.

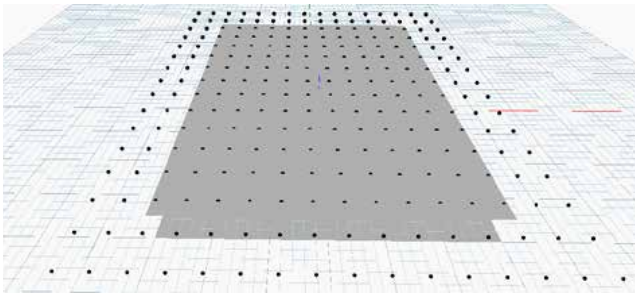


Figura 3.36: Flexibilidad de Estructuración. Se muestra que se puede proponer un grid que tenga dimensiones mayores a la planta de la edificación, de esta manera se da más variabilidad a la configuración que puede ofrecer el estructurista.

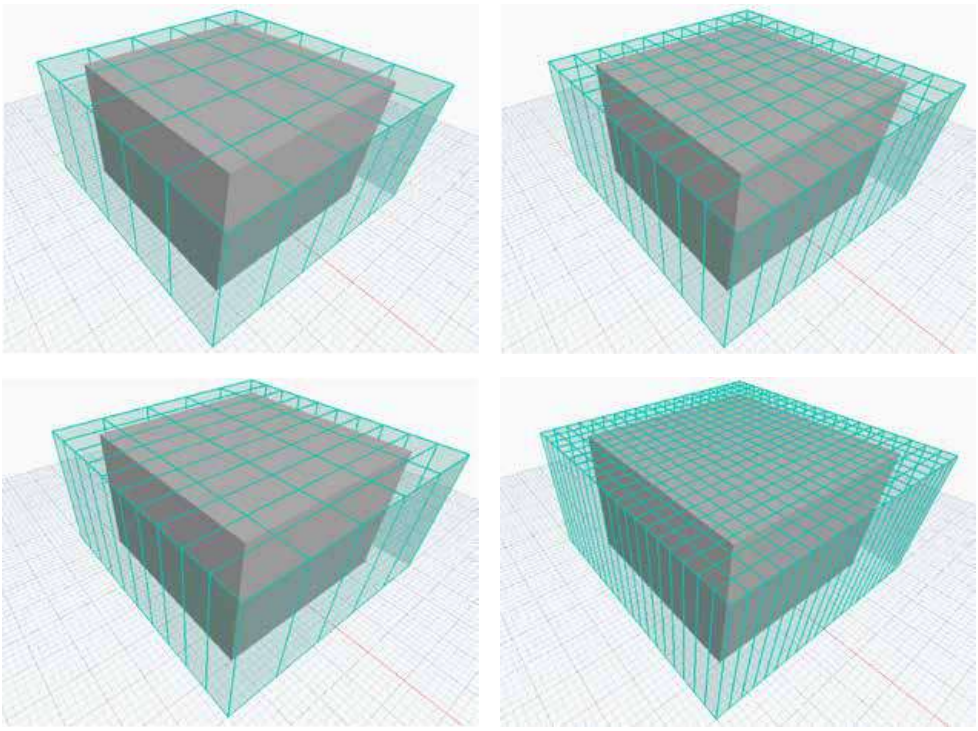


Figura 3.37: Visualización de Grid. Se decidió que la visualización mediante puntos era poco amigable, por lo que se implementó la utilización de una caja con transparencia que permite ver el sólido contenido y líneas guía para definir la posición de las vigas.

información importada desde las primeras conectividades del DAG. Entonces la primer acción del main se da mediante una función dentro de un ciclo que recorre los niveles de la edificación. Aunque se hace una separación en cuanto al primer nivel correspondiente a la base de la edificación, el algoritmo funciona de la misma manera para los demás niveles. La función `get_ret(ret_dim, height, sep_in, ret_lines_g)` comienza definiendo la frontera de la retícula, la cual define el usuario con dos valores que están implícitos en el parámetro *ret_ dim*. Posteriormente se utilizan algunos ciclos para crear puntos desde un extremo de la retícula al otro y usando como pasos las separaciones de los pórticos, por lo que a manera de una matriz se van colocando los puntos que después podrán definir las columnas a estructurar. Después se utiliza un algoritmo de ordenación para conseguir el acomodo con respecto a X y a Z, ya que con ayuda de los puntos es posible crear lineas que definen la retícula en sí. Por lo que teniendo alineados los puntos por ejemplo con respecto a X, se puede simplemente unir por renglones dichos puntos mediante líneas, y de igual manera con la dirección en Y.

Al terminar esta función, se retornan algunos valores como el valor máximo geométrico donde se ubica la frontera de la retícula, las líneas generadas y más importante aún, los puntos donde se ubicaran las columnas en el módulo de estructuración. Al inicio solo se graficaban puntos, pero ahora se generan

superficies perimetrales a la retícula, formando una "caja" donde se deberá introducir el sólido de la edificación, además las líneas superiores a esta caja permiten definir la posición de las vigas, figura 3.38. De esta manera es sencillo proponer la configuración estructural, además de darle un aspecto estético y amigable al procedimiento de introducción de datos.

Aunque al inicio se pensó que este módulo realizara algunas tareas adicionales que son necesarias para la identificación de losas en un procedimiento posterior, se decidió que era mejor separar en dos este módulo, ya que el siguiente paso requiere de mayor costo computacional y al cambiar valores en el grid y activarse las funciones complementarias para losas, la latencia era muy alta e inclusive era tedioso proponer varios modelos. Por lo tanto, este modulo termina en este paso, permitiendo una rapidez considerable al variar los parámetros, es decir que casi en tiempo real se realizan las operaciones descritas anteriormente para que el usuario no demore en definir su configuración para analizar.

3.3.1. Obtención de Superficies para Losas

El módulo ***Grid*** fue planeado para realizar diversas funciones en donde es importante su interacción para que el usuario pueda definir elementos como lo son las barras en la estructuración y ayuden en la creación de losas de cada nivel, las cuales se revisarán con más detalle más adelante. Sin embargo, dada la etapa de este módulo fue más conveniente obtener cierta información necesaria para losas desde este punto, para así no tener que hacer cálculos innecesarios nuevamente en un proceso posterior.

La cuestión es que al implementar las funciones complementarias para losas, el tiempo de cálculo se incrementó mucho en este módulo y la utilización de la aplicación completa pudiera no ser eficiente para el usuario ya que había que esperar tiempo considerable solo para la generación de la retícula de ayuda. Esto quiere decir que para generar muchos modelos, era muy tedioso cambiar un parámetro y actualizar. Por ello se pensó en disminuir el tiempo de operabilidad partiendo en dos módulos el código, el primero es el explicado al inicio de este capítulo y el cual se buscó fuera lo más "ligero" posible, para que tuviera un flujo más continuo el realizar cambios de la estructuración que asigna el usuario, es así que éste tiene una latencia de fracciones de segundo. Por otro lado el segundo módulo creado tiene un costo computacional alto, por lo que este se incluirá al momento de generar la estructura completa, ya que aquí el usuario no tiene una interacción directa, sino que al ejecutarse solamente tiene que esperar a que termine el programa. El código de este módulo se encuentra en el apéndice A.4.

El objetivo de este módulo es obtener las superficies que serán incorporadas al módulo de distribución de cargas en losas. Para ello la metodología que aquí se explica tuvo que calibrarse hasta un punto en donde se obtuvo mayor flexibilidad para edificaciones con plantas irregulares. Esto se menciona debido a que hasta este punto pudiera resultar un poco ilógico el método empleado, ya que necesita de cálculos con geometrías más complejas como lo son superficies y sólidos, cuando los resultados del módulo anterior tienen definida la geometría de cada losa mediante puntos y líneas. Esto se explicará en su momento, pero hasta ahora solo se abordará la explicación de este módulo y como es que se consiguen las superficies.

El primer paso es recibir la información generada por el módulo anterior, la cual será necesaria para las funciones aquí programadas. Posterior a esto y dentro de un ciclo que recorre cada nivel de la edificación, incluyendo la azotea, se activa la función `get_slab_ret(ret_lines,sep,floor_surf)` que tiene como meta obtener las superficies que fungirán como losas. La idea general es que a partir de la retícula que el usuario observa, no solo se obtienen puntos sino que además se generaron líneas. Éstas líneas son acomodadas de tal forma que en el primer paso de esta función es organizarlas por grupos de cuatro líneas y al darles dirección pueden formar rectángulos, y con ayuda de una función de **Dynamo**, es posible generar una superficie a partir de líneas conectadas y que formen un polígono cerrado. Esta función propia de la librería de geometría utilizada, se denomina `Surface.ByPatch(Polycurve)` y solo se necesita de una policurva que en este caso se refiere al rectángulo creado antes. Teniendo la superficie rectangular correspondiente al *grid*, el siguiente paso es hacer una intersección entre la superficie del nivel completo con esta, usando la función `Geometry_1.Intersect(Geometry_2)`, donde Geometry_1 se refiere a la superficie del nivel y Geometry_2 es la superficie rectangular del Grid. La pregunta ahora pudiera ser: ¿Porqué hacer esa intersección, si ya tenemos la geometría de la losa al utilizar el rectángulo y la función `Surface.ByPatch(Polycurve)`? Pues la respuesta se debe a que si solamente nos interesa analizar edificaciones rectangulares, este procedimiento no tiene sentido y solo le suma latencia no deseada. Pero si queremos añadir flexibilidad a la variabilidad de edificaciones, como por ejemplo, un edificio con forma circular o irregular (ver imagen 3.38), las losas en los extremos seguro tendrán una forma muy distinta al rectángulo que se genera mediante el grid. Por ello se consideró esta estrategia que permite obtener cada superficie de losa con la forma que sea, sin embargo como se ha comentado la operación de creación e intersección de superficies tiene un costo computacional elevado, por ello se implementó un código basado en programación en paralelo (véase sección 3.2.1), pero en lugar de separar el número de niveles, se hizo una división de regiones de acuerdo a las superficies del grid que serán utilizadas para la intersección con la superficie completa del nivel. A diferencia de la implementación de programación en paralelo en la obtención de niveles del módulo ***Floors***, esta vez

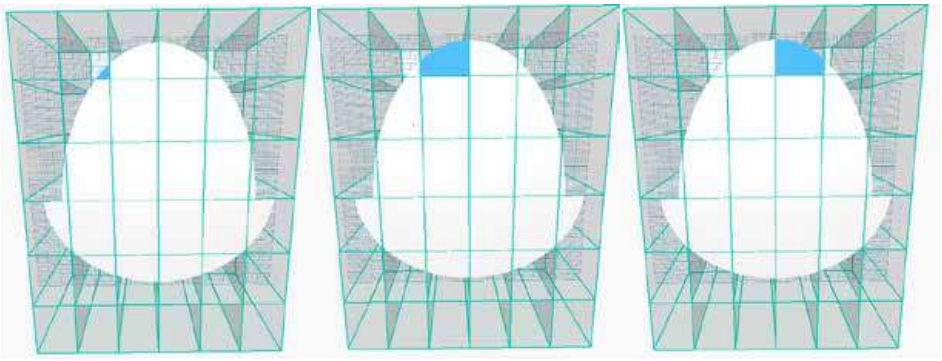


Figura 3.38: Obtención de Superficies. Edificación con una planta no rectangular, se muestra en color azul tres ejemplos de resultados por la intersección de superficie de nivel completo con la superficie generada por cada rectángulo en la retícula.

se comprobó la efectividad del método con una diferencia aún mayor, debido a que la reducción de latencia estuvo fuertemente relacionada con el cálculo simultáneo de cada región. Tal como se menciona antes, el procedimiento para la codificación en paralelo consta de tres etapas principales, la primera es la separación en regiones de cálculo, la segunda se refiere a las instrucciones que se ejecutarán simultáneamente y la tercera se refiere al hecho de volver a conectar las regiones separadas para garantizar la continuidad de los cálculos. La imagen 3.39 muestra de manera sencilla los pasos que se emplearon para la programación en paralelo.

Para este módulo también se hicieron algunas pruebas de medición de latencia y se encontró que para un número pequeño de superficies el algoritmo paralelizado se vuelve más lento que el original, es por ello que también se le asignó un mínimo de superficies para que se activará la división de regiones.

3.3.2. Cálculo de Centroides para Losas Poligonales

Por lo pronto se generó una superficie por cada losa, la cual nos va a servir posteriormente para el capítulo de distribución de cargas. Un aspecto importante en la distribución de cargas mediante métodos geométricos, como los utilizados en este trabajo, es que la manera de asignar las fuerzas a las barras que soportan las losas depende de la forma de la misma losa, esto se explicará mejor en el capítulo correspondiente. Pero por ahora se mencionará que en losas rectangulares la distribución se hace mediante triángulos y trapecios, pero en losas poligonales se deberá calcular el centroide y la distribución se realizará mediante las áreas resultantes. Y aunque existen diversos métodos para realizar esta labor, se decidió probar una función propia de **Dynamo**. El problema aquí es que esta función sirve únicamente para sólidos, no para superficies. Entonces, se programó una función denominada `get_solids(surfaces)` la cual recibe un vector de superficies y genera un sólido utilizando una función de la librería de geometría, `Surface.Thicken(Int, Bool)`, la cual está dentro de un ciclo que recorre todas las superficies. El sólido se genera con un espesor

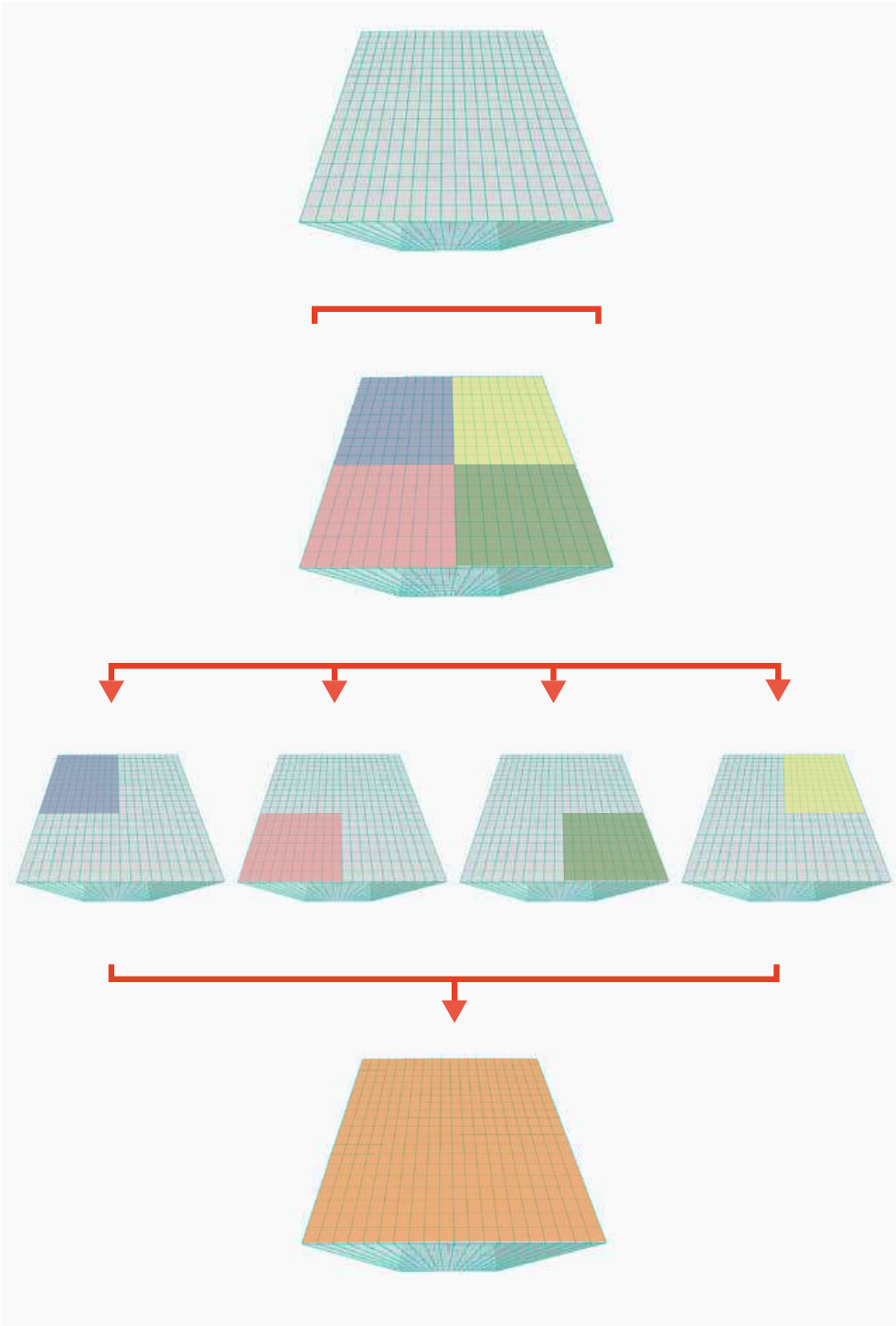


Figura 3.39: Paralelización de Intersecciones. Explicación de la estrategia de programación en paralelo utilizada. Se muestra la división de regiones, el cálculo optimizado para cuatro núcleos y la conexión de regiones para permitir la integridad de la planta.

pequeño, solamente representativo para crearlo, ya que las coordenadas que nos interesan son las contenidas en el plano $X-Y$. Ahora se procede a utilizar la función `Solid.Centroid()` la cual regresa un punto con las coordenadas del centroide. Los pasos descritos antes se encuentran inmersos dentro del concepto de programación en paralelo, por ello para llamar la función que genera sólidos y posteriormente centroides se tuvo que dividir en bloques iguales el número de losas a considerar, esto se realiza como se explicó an la sección anterior. Primero se divide, posteriormente se llama a las funciones de la librería *Threading*, en donde por ahora optimizamos a cuatro hilos. El resultado de este segmento de código es un arreglo de puntos geométricos que denotan los centroides por cada losa a considerar. Sin embargo por el uso de la paralelización los puntos están desordenados por toda la planta, sin ninguna lógica; cómo se explicó antes, los núcleos de la computadora al estar trabajando simultáneamente, estarán "peleando" la memoria a utilizar y por ello "aleatoriamente" regresaran los resultados sin algún orden. Se tuvo que implementar un algoritmo de ordenación matricial de acuerdo a nuestras necesidades. Se necesita saber que posición del arreglo contiene a un centroide que coincida con las losas. Primero se guardan sub-vectores que contienen centroides que comparten la misma coordenada en X, de esta manera se tiene la primer ordenación respecto al eje X. Pero cada vector esta desorganizado en cuanto al eje Y, por lo que se utiliza un ciclo anidado el cual recorre cada sub-vector y acomoda los centroides de acuerdo a su coordenada en Y de manera creciente, negativo a positivo. De esta manera ya se tiene conocimiento del lugar que ocupa cada centroide en el arreglo y por ende se puede hacer uso de éstos junto con las superficies obtenidas en la intersección anterior que fungirán como losas.

En el momento que las funciones enlistadas en la región denominada "Main" han terminado, un paso muy importante es liberar la memoria correspondiente a los elementos geométricos intermedios utilizados para obtener los resultados esperados. Este procedimiento no es común al utilizar *Python*, sin embargo a continuación se explicará el motivo.

3.3.3. Liberación de Memoria en Python

Los programadores de lenguajes como C o C++ sabrán que al utilizar variables que necesiten una cantidad de memoria considerable, la mejor práctica de programación es hacer una liberación de ésta al final del programa para evitar comportamientos no deseados durante la ejecución del código, e incluso prevenir errores que pueden ser graves. Sin embargo es bien sabido que la programación en lenguajes de alto nivel como lo es python *"no requiere de atención especial"* en cuestiones de memoria ya que cuenta con un sistema de gestión automática o implícita. Esto es un avance muy importante, ya que el hecho de que no debamos preocuparnos por cuestiones técnicas como la memoria y centrarnos más en el fin de la aplicación que estamos desarrollando permite

tener un avance más eficiente. En el caso de Python el mecanismo de gestión implícita se denomina *"Garbage Collector (Recolector de Basura)"* y funciona de la siguiente manera:

- Al crearse una nueva variable (durante la ejecución del programa), la cual puede contener un número, string, caracter, o algún otro, se reserva un sector de memoria para ser almacenado. No es necesario que el programador tenga que saber el tamaño específico de dicha variable, ya que el compilador se encarga de identificarla.
- Las variables en Python hacen referencia a objetos que a su vez regresan el valor de lo que está contenido, no es necesario acceder a la dirección de memoria creada por la computadora.
- Cuando se deja de referenciar un objeto, es detectado por el *Garbage Collector* quien será el encargado de liberar la región de memoria utilizada.

Muchos lenguajes modernos (además de Python) tienen un sistema similar de manejo de memoria, algunos aspectos generales que se deben tener en cuenta son los siguientes:

Para que el garbage collector pueda hacer su trabajo, se requiere de contadores de referencias para cada sector de memoria. Para esto colaboran el compilador y el entorno de memoria. Realmente a nosotros como programadores de estos lenguajes no es muy importante como es que se definen y almacenan esas referencias, sino más bien que relaciones o reglas tienen para saber diferenciar entre un objeto referenciado y uno que deja de serlo. Estas convenciones dependen del lenguaje utilizado.

Un objeto está referenciado si existe una variable activa que lo esté apuntando. Un ejemplo sencillo es el siguiente: si tenemos una función donde se define una variable local "`hora`", al ejecutar la función y terminar, las variables locales dejan de existir. Sin embargo el retorno de esa función (la fecha que solicitamos) apunta a la referencia de la variable local por lo que ¡aún sigue existiendo! Lo que sucederá es que cuando la variable deje de referenciarla, en algún momento el *Garbage Collector* va a detectar que la fecha tiene estatus de no referenciada y va a liberar su memoria (11).

En pocas palabras, el Garbage Collector de Python consiste en:

- Un Subproceso del programa que periódicamente estará recorriendo la memoria, y analizando si cada sector tiene referencias o no.
- Cuando encuentra un sector no referenciado, lo libera.
- Además se encarga de optimizar el uso de la memoria, especialmente su fragmentación.

Entonces, esta herramienta que proveen los lenguajes de alto nivel como Python permite una construcción de código más enfocada al propósito del mismo y sin preocupaciones de la gestión de memoria de la computadora. Sin duda alguna es una de las mayores ventajas de la implementación de herramientas en estos lenguajes, pero si estamos hablando de tantas ventajas al usar el *Garbage Collector* ¿Por qué dijimos antes que un paso muy importante al final del código del cálculo de centroides era liberar la memoria utilizada?.

3.3.4. ¿Porqué liberar memoria si estamos usando Python?

Se acaba de mencionar que los lenguajes de alto nivel como lo es Python, tienen herramientas avanzadas de gestión de memoria (*Garbage Collector* en el caso de este lenguaje). Entonces, ¿Cuál fue el motivo de gestionar la memoria manualmente? Bueno, pues es importante mencionar que esta sección en realidad no fue planeada para involucrarse en este trabajo de tesis, la utilización de la liberación de memoria fue un obstáculo que surgió mientras se programaban las funciones principales, sobre todo al involucrar la función de obtención de centroides.

¿Cómo se originó? Pues ya se tenía un planteamiento para la generación de la estructura y la identificación de losas para la distribución de cargas, este era un código muy "ligero" ya que la metodología constaba solamente en el uso de puntos. Pero la limitante principal de este código era que solamente funcionaba para losas rectangulares, por lo que las únicas edificaciones que se podían modelar eran prismas cuadrados o rectangulares. Por motivo de dar flexibilidad al código se optó por utilizar el método de intersección de superficies rectangulares del módulo ***Grid*** con la superficie del nivel completo obtenida en el módulo ***Floors*** explicada en las secciones 3.3.1 y A.2 respectivamente. Al momento de programar la intersección de superficies, se estaba modelando una edificación sencilla de un nivel como prueba inicial, con no más de unas 10 losas. Hasta ese punto todo funcionaba bien, se prosiguió programando módulos subsecuentes y al momento de hacer pruebas con edificaciones más grandes, con más losas y con más pisos, el software de modelado **Dynamo** empezó a comportarse de manera inestable. Algunas veces se corría el código y todo funcionaba bien, pero algunas veces no ocurría esto. En los mejores casos, aparecía un mensaje "Warning" en los módulos, al igual que cuando se comete algún error simple. Pero algunas otras veces, al momento de ejecutar el programa, simplemente se cerraba el modelador, un comportamiento completamente inesperado. Algunas veces, creí tener suerte porque este comportamiento no pasaba más de un par de veces por día. Otras ocasiones, cada vez que corría el código se cerraba la ventana del modelador. Debo confesar que esta etapa fue muy frustrante, no tenía una sola idea de lo que ocurría. Al encontrarme codificando módulos posteriores al de ***Grid***, siempre pensé que en esos puntos estaba el error, pero no lograba identificarlo. Desde fallas en los algoritmos programados, fallas en la

Figura 3.40: !Peligro! Representación del error en el manual de **Dynamo** en GitHub sobre gestión de memoria en geometrías de dimensiones altas (14).

licencia del modelador y hasta fallas en mi propio ordenador eran planteamientos que me hacía acerca de la inestabilidad en el funcionamiento de **Dynamo**. La investigación en línea fue pesada, y no fue hasta que en un foro ((14)) publicaron una respuesta que ligaba a un manual alternativo de la programación sobre el modelador. La respuesta a mis problemas estaba representada en el manual con la imagen 3.40 que difícilmente olvidaré por mucho tiempo.

Literalmente el título en la imagen del robot dice: "¡Si no implementas esto, pondrás a **Dynamo** y a **Revit** en un estado indefinido, causando que ambos fallen misteriosamente!" (18). Por lo que el uso de las sentencias `Dispose` o `using` son recomendadas al utilizar objetos geométricos que necesitan mayor memoria. El manual menciona que si estas empleando estas geometrías de manera intermedia (como herramienta) para generar otra geometría final que será retornada en tu código, necesitarás liberar manualmente la memoria. Esto ocurre exactamente en el código de ***Grid***, ya que estamos empleando las superficies rectangulares en un ciclo para hacer intersecciones en una superficie de nivel. Además, para el calculo de los centroides usamos estas superficies para convertirlas en sólidos, aplicamos la función `Solid.Centroid()` y así obtenemos los centroides. En este módulo prácticamente lo único que necesitamos retornar de las funciones es un arreglo de puntos (centroides) y un arreglo de superficies (losas), pero para ello ya utilizamos líneas, superficies e incluso sólidos como geometrías intermedias. Por lo que al confiar en el *Garbage Collector* de Python para liberar automáticamente estas referencias no era buena idea, ya que se ocasionaba fallas inesperadas.

Me pareció importante incluir esto en el trabajo de tesis, ya que me costó muchas semanas encontrar el error y pienso puede ser de gran ayuda para la

persona que quiera escribir código en cualquier modelador paramétrico. Entonces incluyendo unas cuantas líneas más de liberación de memoria en geometrías altas como superficies y sólidos, el modelador se comportó de manera adecuada y no tuve que volver a preocuparme por este error. Tal como dice el manual de GitHub, esto puede ocurrir dentro de **Dynamo** como pasó en mi caso, pero además puede ser un error simultáneo con programas vinculados como **Revit** y usando otros lenguajes además de Python como puede ser "C#". Simplemente hay que buscar la sentencia que libera la memoria en cada caso.

3.4. Propiedades de los Materiales y Secciones Transversales

El análisis matricial estructural requiere de diversa información para realizarse, tal como se revisará en la sección 3.7 de este capítulo. Cómo ya sabemos, se necesita la idealización de la configuración estructural, la topología o conexión de los elementos estructurales, las condiciones de desplazamiento permitidas y las restringidas, las fuerzas aplicadas pero además las características geométricas de los elementos estructurales y las propiedades de los materiales que están compuestos. Por ello es necesario una entrada de datos para indicar al programa de análisis que tipo de sección transversal se está usando, las propiedades geométricas necesarias para el análisis[1] son:

- El área de la sección transversal.
- Momento de inercia respecto al eje X
- Momento de inercia respecto al eje Y (3D)
- Momento de inercia polar J para considerar torsión (3D)

Es claro que se debe asignar cada propiedad geométrica correspondiente según la barra, por ello se debe relacionar el índice de ésta con la sección transversal seleccionada. Además se deben agregar propiedades mecánicas de los materiales empleados, las cuales son usadas para el cálculo de las rigideces que componen las matrices para el análisis, dichas propiedades necesarias son:

- El módulo E de elasticidad.
- El módulo de corte G.

[1] Estas propiedades son las necesarias solamente para el análisis estructural mediante el método de las rigideces, si se requiere revisar la resistencia del modelo, es necesario involucrar otras propiedades según el tipo de material (Concreto, Acero Rolado en Caliente, Acero Rolado en Frío, Madera, etc.)

Aunque también otras propiedades del material pueden ser útiles, no son indispensables para realizar el análisis estructural. Como por ejemplo el peso específico γ_{mat} que se utiliza como fuerza gravitacional adicional a las que el usuario asigna, o la relación de Poisson ν para la deformación tangencial y longitudinal, el cual también es un parámetro para obtener la relación entre módulos de elasticidad normal y cortante, o el factor térmico utilizado en análisis que involucran cambios en la temperatura. Incluso si se desea verificar la resistencia del material mediante un código de diseño muy seguramente se pedirán otros datos específicos de cada material como lo son la resistencia a compresión del concreto $f'c$, o el límite de fluencia en el acero f_y, entre otros. Pero tal como se comentó antes, los datos necesarios para el análisis estructural solamente son los que se encuentran en lista.

Entonces, se necesita de un módulo donde se ingrese esta información al modelo estructural y además éste debe tener contacto directo con el usuario ya que de esta manera se podrá elegir el tipo de barras que se emplearán. Una primera opción, quizá la más sencilla es que el usuario proporcione cada propiedad tanto geométrica como mecánica a cada barra - como en los software de análisis estructural comerciales - sin embargo estamos intentando abordar una reducción de tiempo en la realización del modelo, y tener la alternativa de optimizar de manera más sencilla. Por lo que el planteamiento utilizado es mediante una característica que posee el software de análisis estructural utilizado **MECA** (del cual se darán detalles más adelante), que además de dar la posibilidad de ingresar los datos exactos de cada propiedad, cuenta con catálogos de materiales y secciones que le permiten al usuario acceder más rápidamente a estos valores. Y aunado a esto, el uso de catálogos da la posibilidad de hacer una optimización estructural que el software tiene implementado mediante un método denominado "Recocido Simulado".

Ya que esta herramienta de parametrización esta pensada que funcione en una etapa de planeación temprana, se pensó que una buena idea es que las propiedades tanto de vigas y columnas puedan modificarse por nivel de la edificación. Por ejemplo si tenemos un edificio de 4 pisos podemos elegir que las columnas del primer al segundo piso tengan una $sección_{C_1}$ y las columnas de los niveles 3 y 4 tengan una $sección_{C_2}$. Pero que las vigas tengan una $sección_{V_1}$ en todo el edificio. Esto permite ingresar la información de manera más rápida, además que en la práctica es común que se tomen soluciones que no varían demasiado en el mismo nivel. E incluso, muchas veces se utiliza una misma sección para todas las columnas y otra para vigas, debido a que se busca estandarizar las secciones. Esto beneficia al modelo paramétrico, que aunque no es imposible seleccionar distintos materiales y secciones en el mismo nivel, casi particularmente por cada viga, requiere de una flexibilidad mayor en el código, lo único que hará es aumentar el número de parámetros y por lo tanto hacer más lento el modelado.

Al utilizar catálogos que brinden la información al modelo estructural, el usuario solamente tiene que escoger el material y la sección a utilizar. Por lo que la implementación en un código fue muy sencilla, se programó un nodo de **Dynamo** el cual solo hace una lectura al catálogo de secciones y devuelve las propiedades necesarias para la entrada de datos del software **MECA**, los códigos se encuentran en los apéndices A.5.1, A.5.2 y A.5.3. Los parámetros de entrada son tres:

1. Número de sección
2. Nivel inicial
3. Nivel final

Se creó un módulo por cada material aceptado por **MECA**, los cuales son:

- Concreto reforzado - CONCR
- Acero rolado en caliente - RC
- Acero rolado en frío - RF

El usuario tiene que seleccionar el nodo categorizado por los materiales anteriores, conectará los tres enteros como parámetros entrada y obtendrá la información a conectar en el módulo de estructuración. Deberá tener por lo menos un nodo por columna y otro nodo por viga, es decir, que en el modelo se deberán incluir por lo menos dos materiales. Aunque esta información aún no será necesaria en este momento, se planeó explicarla en este segmento ya que el orden de este trabajo escrito se basa en el recorrido que debe realizar el gráfico *DAG* desde el primer parámetro hasta el resultado final.

3.5. Generador de Geometría Estructural

Aunque todos los módulos que se programaron para este trabajo son indispensables para obtener los resultados de un análisis estructural mediante geometría paramétrica, quizá el que se encuentra en el centro de todo es este, denominado ***Structure***. El objetivo de esta tesis es obtener de manera sencilla la información necesaria para analizar una estructura, que fuera amigable y a su vez redujera el tiempo de modificaciones. Entre las opciones que se plantearon, la idea de utilizar un sistema paramétrico fue la que pareció innovadora y con un enfoque ideal para el trabajo que se realiza en Ingeniería Civil. Entonces, de cualquier manera que fuera la interacción con el usuario o el traspaso de información al software de análisis, el objetivo primordial era obtener una estructuración, y este módulo se centra en esa parte. A través del tiempo que se comenzó a investigar el funcionamiento de la parametrización, se buscaron

alternativas para que el usuario pudiera definir la configuración estructural, por lo que este código ha sido el que más ha cambiado a lo largo de la realización del trabajo.

3.5.1. Structure Generator

Sin duda una de las referencias principales para realizar este trabajo se debe al software **Structure Generator**. Este software se encontró de casualidad al investigar sobre parametrización, y lo más curioso fue que lo único que se pudo conocer de éste son unos videos en francés que explican el funcionamiento. Al parecer esta idea surgió en una incubadora de ideas que tiene la compañía "Autodesk", no se pudieron encontrar pruebas de su comercialización o de su utilización por el público en general, solamente esos videos extraordinarios que muestran un modelador tridimensional, que al tener un modelo definido y ejecutar el programa, pareciera que por "arte de magia" los elementos estructurales aparecieran y fueran modificados de manera muy sencilla, además este programa estaba vinculado con el software de análisis estructural que ofrece esta compañía, denominado "Robot". Aún cuando se buscó la forma de conocer más a fondo de este programa, el intento fue fallido y solamente sirvió como una influencia muy grande en el proyecto a realizar. Aunque los desarrolladores de esta aplicación, que claramente fueron empleados de Autodesk, denotaban un trabajo excepcional tanto en la complejidad de los cálculos que realiza el programa, así como de la interfaz gráfica y el visualizador incluido, por alguna razón este proyecto fue detenido. Entonces este ha sido una motivación para dar los primeros pasos en el análisis de edificaciones mediante parametrización geométrica. La figura 3.42 denota la primera parte del video en donde se traza un modelo tridimensional que se pretende estructurar. La forma de trazarlo es mediante una herramienta CAD que al parecer tiene funciones que simplifican este proceso. Se traza cada planta para formar las superficies que varían para finalizar creando el sólido 3D.

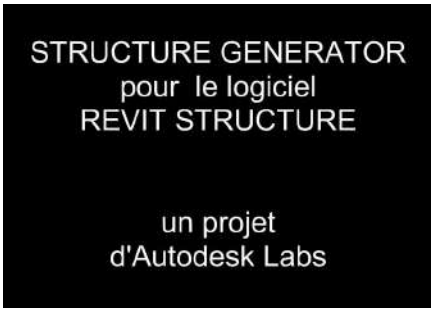


Figura 3.41: Structure Generator. Video (francés) encontrado por casualidad en internet donde se estructura un ejemplo de edificación, no se encontró mas material además de estas imágenes

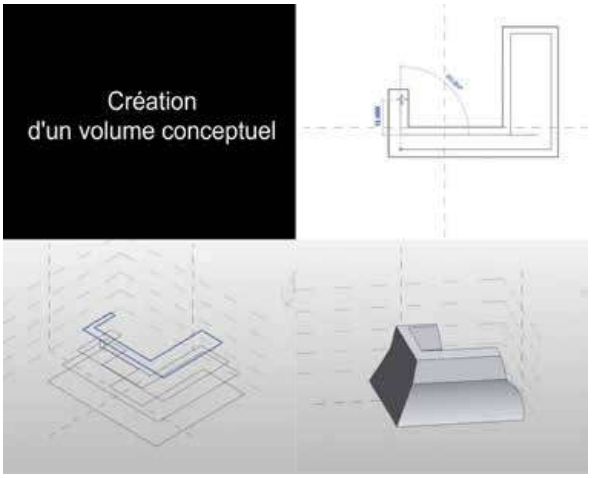


Figura 3.42: Structure Generator. Funcionamiento del software experimental, en estas primera etapa se traza un modelo tridimensional en un sistema CAD.

Al haber realizado un modelo tridimensional en un CAD complementario, se ejecuta **Structure Generator** y aparece una ventana ya con el sólido aparentemente con una malla que define la geometría externa como se puede observar en la figura 3.43. Se puede observar una sección inferior con opciones de estructuración, y apreciando solamente el video se observa que el programa secciona por pequeños rectángulos el sólido, de forma similar a un programa de pre-proceso de elementos finitos. Y en la tabla se alcanza a observar el número de cada elemento (rectángulo) y pregunta como estructurarlo. Es decir, que cada rectángulo funciona como "nicho", entonces puedes elegir si quieres columnas izquierda y derecha como un marco tradicional, o si gustas colocar contravientos inclinados, entre otras opciones que se ven en el acercamiento de la figura 3.44. El siguiente paso que aparece en el video demostrativo es el uso de los parámetros para cambiar la estructuración y tener distintas propuestas.

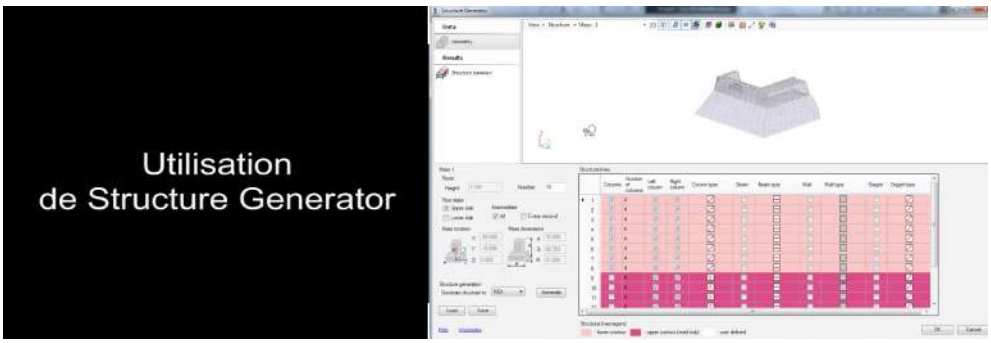


Figura 3.43: Structure Generator. Se ejecuta **Structure Generator** y aparece una ventana con el sólido y opciones de estructuración.

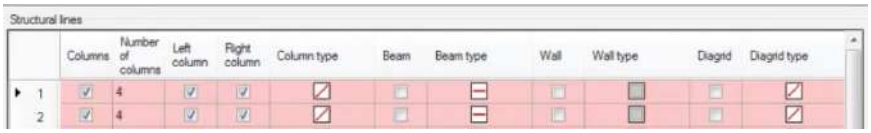


Figura 3.44: Structure Generator. Detalle de la tabla que contiene cada elemento rectangular y las opciones para estructurarlo.

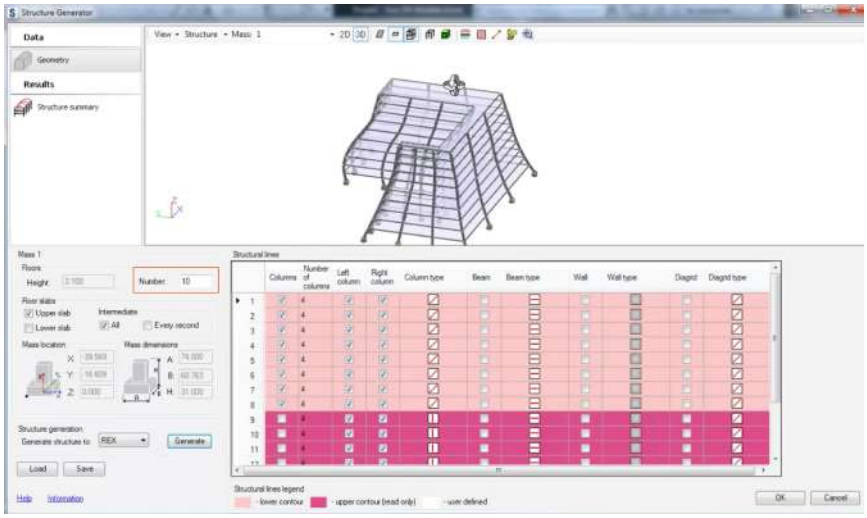


Figura 3.45: Structure Generator. Estructuración del sólido tridimensional, se observa que se puede modificar algunos parámetros como lo es el número de niveles, entre otros.

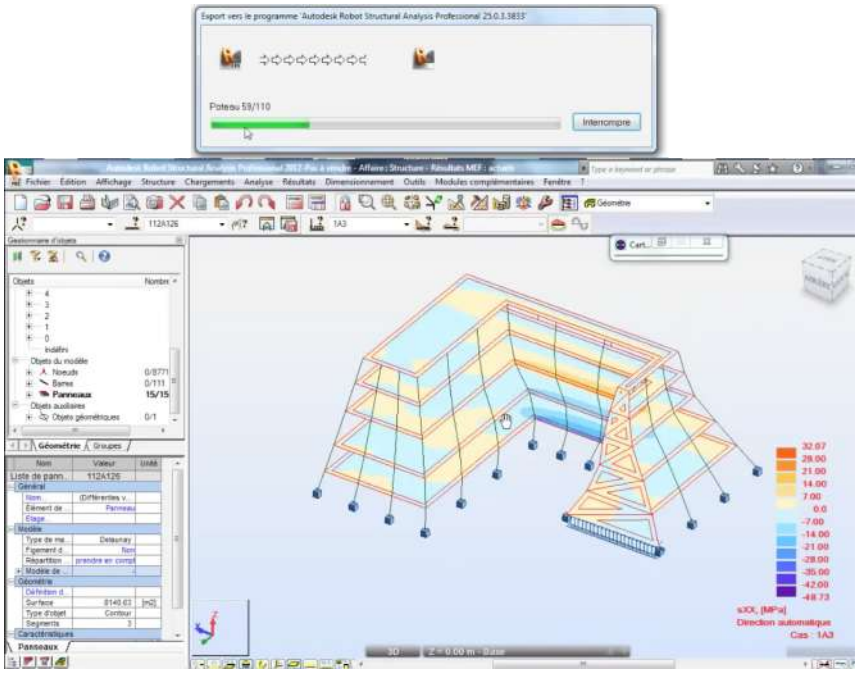


Figura 3.46: Structure Generator. Exportación de la geometría e información estructural al software de análisis estructural **Robot**.

Algo que es importante en este software experimental, es que además de generar una configuración estructural en base de geometría tridimensional es posible realizar análisis utilizando software vinculado. Esto es muy adecuado, ya que como se mencionó antes, este programa se realizó en los laboratorios de *Autodesk*, donde tienen una gama muy amplia de software especializado. Por ello tienen oportunidad de agregar complementos que permiten tener un control de la información más eficiente. **Structure Generator** tiene una opción de exportación a un software de análisis estructural denominado **Robot**. La imagen 3.46 muestra esta fase de exportación y una ventana de este software de análisis donde se muestra ya el modelo estructural idealizado. Algunos de los ejemplos de estructuración que este software presenta son los mostrados en las imágenes de la figura 3.47. Algo que destacar es que se muestra las tres principales etapas de estructuración, primero se tiene un sólido conceptual tridimensional que es sometido a un algoritmo de mallado para conocer sus límites geométricos. Posteriormente se genera el modelo idealizado estructural, mediante líneas simples para en la última imagen mostrar un render del edificio con las secciones transversales utilizadas, lo que le da un aspecto estético muy bueno.

Desafortunadamente no hubo oportunidad de probar este software de generación estructural, por ello simplemente sirvió como referencia visual de lo que se puede lograr utilizando de forma correcta el poder de la computación en la ingeniería. El arma utilizada en este trabajo de tesis fue la utilización de parametrización para intentar reducir el tiempo en la creación de la geometría, preparar la información y realizar un análisis estructural. Muy similar al objetivo que tuvo Structure Generator.

3.5.2. Estructuración Externa Mediante Intersección de Niveles

Cuando se decidió utilizar sólidos tridimensionales, como materia prima para alimentar al algoritmo, el primer planteamiento fue en identificar los límites de cualquier sólido presentado. Por lo que se implementaron funciones que arrojaban lluvias de puntos, los cuales al unirlos funcionaban como una malla que describían los elementos estructurales deseados. No nos interesaba conocer la geometría de una edificación por completo, como sería necesaria en un análisis de elementos finitos por ejemplo, sino que solamente nos interesa conectar nodos que vayan del piso inferior al piso superior (columnas), y nodos que estén contenidos en los planos que representan los niveles con nodos con un patrón regular (vigas), que es como se puede estructurar una edificación regular.

Tal como se explica en la sección 3.3, al tener la intersección de planos con el sólido tridimensional se obtienen los niveles, entonces nuestra "malla"

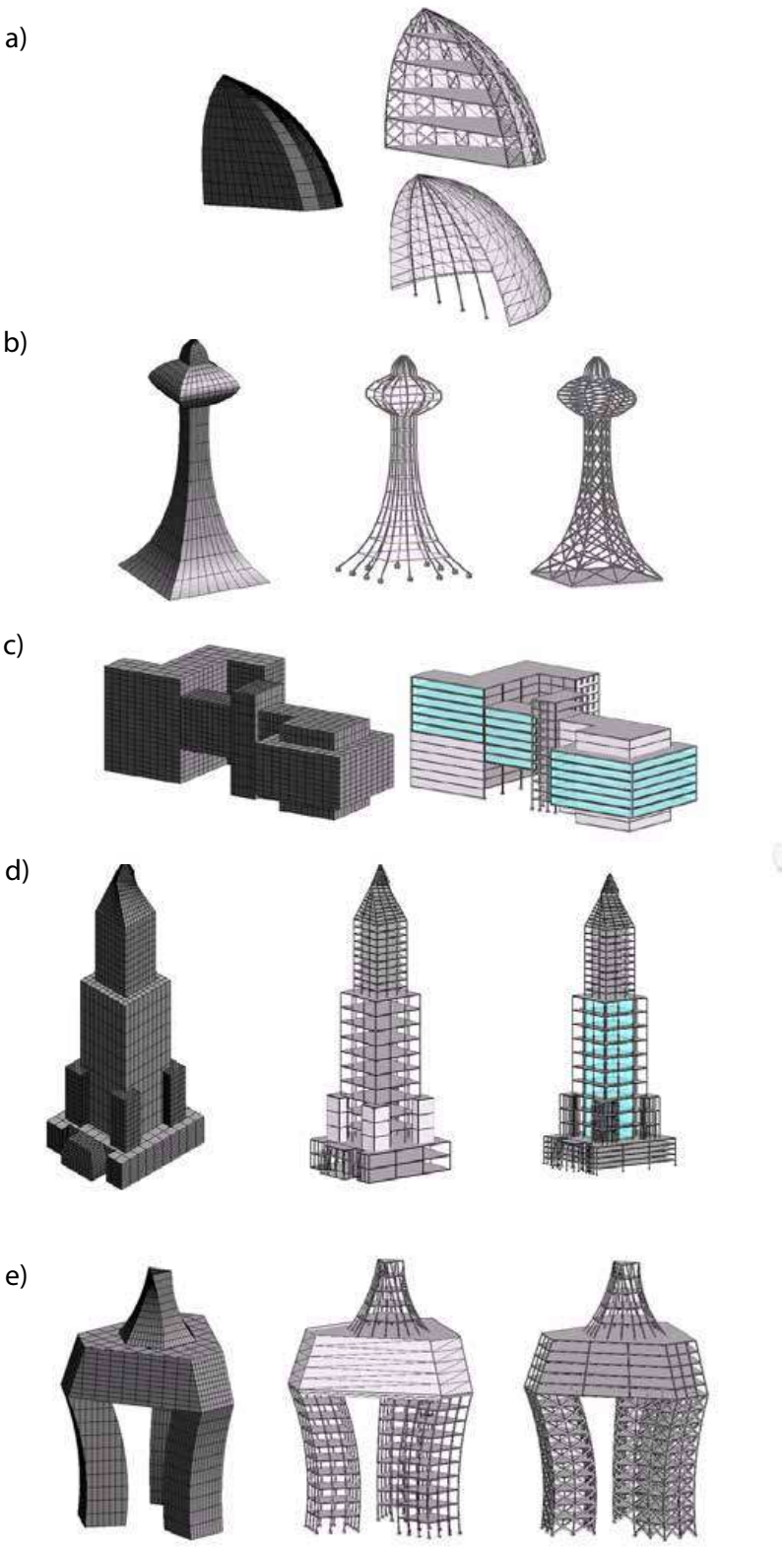


Figura 3.47: Resultados de Structure Generator.
a) Segmento curvo de lo que parece ser una bóveda o un recinto como tipo auditorio, por lo que se denota una gran flexibilidad en los parámetros estructurales.

b) Modelo de una torre que es estructurada de manera que los elementos curvos se pueden proponer como armaduras, se observan las tres etapas de estructuración.

c) Edificación de varios niveles, que denota una arquitectura tipo minimalista, lo interesante es que se puede estructurar aún cuando tiene huecos en el sólido.

d) Modelo de una edificación alta o rascacielos, que muestra algunas geometrías complicadas como la parte superior que parece dar un giro, este tipo de sólidos son lo que interesan para el desarrollo de esta tesis.

e) Se muestra una edificación atípica con tres grandes segmentos para sostenerla, este modelo aunque se pareciera no tener lógica para una construcción, el software es capaz de proponerlo y analizarlo.

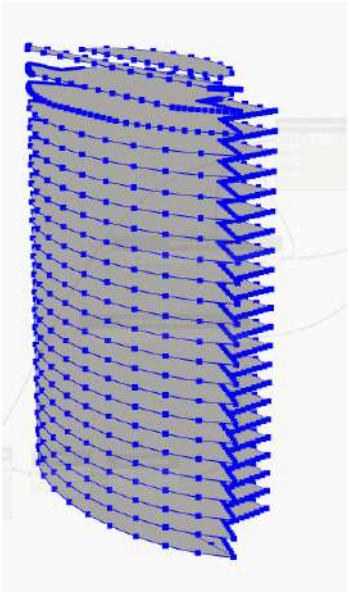


Figura 3.48: División de Curvas Perimetrales. Posibles nodos a utilizar sobre el perímetro de cada nivel en el modelo de la Torre Mayor.

hasta ahora tiene esa información, los puntos que inicialmente se añadieron con el primer algoritmo, se organizaban en torno a los perímetros de dichas superficies, el número de puntos era el parámetro que el usuario brindaba, es decir, el número de columnas que rodean la edificación. Entonces el primer algoritmo buscaba generar la estructura exterior, muy similar a lo que planteó **Structure Generator**.

El primer algoritmo dividía las curvas de cada perímetro en los niveles de acuerdo al número de separaciones que el usuario deseaba, sin embargo, para sólidos compuestos como vemos en la figura 3.48, cuando una superficie estaba rodeada de distintas curvas, el usuario debía saber cuantos parámetros utilizar (un número entero por cada curva) y además al tener distintas divisiones en un mismo perímetro, es muy complicado unir elementos estructurales como columnas. Podemos ver en la misma imagen que en los pisos superiores donde la geometría tiene un chaflán, la concentración de puntos es mayor que en los primeros niveles, entonces obtendríamos columnas que tendrían distintas inclinaciones en cada nivel y por ende es aún más complicado construir.

El planteamiento en base a división de curvas perimetrales no es muy lógico en cuestión constructiva. Por ello se pensó que debía ser de tal manera que el usuario definiera el número de columnas principales desde la primer planta y que éste siguiera en la totalidad del edificio. Esto tiene más lógica ya que en la estructuración de edificaciones con aspecto mas o menos regular, se tienen elementos estructurales que desde la cimentación dan continuidad hasta el último piso, y en base a estos se hacen las conexiones de vigas principales y secundarias. Entonces, el procedimiento comienza cuando el usuario define el

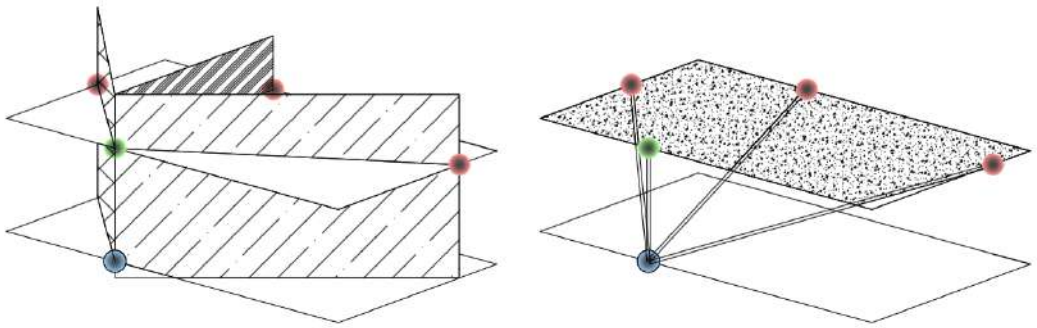


Figura 3.49: Metodología para estructuración externa. Procedimiento de intersección de planos con el nivel superior para obtención de posibles puntos o nodos finales por columna, en la figura derecha se muestran los candidatos a columnas de acuerdo a los puntos obtenidos con las superficies.

número de columnas por cada "cara" que tiene la edificación. Se generan unos puntos en la base, de los cuales el algoritmo comenzará a generar las columnas en base a intersecciones de planos verticales con cada nivel de la edificación. La figura 3.49 muestra este procedimiento.

Tal como observamos en la imagen izquierda de la figura 3.49, tenemos una superficie inferior (nivel 0) y una superficie superior (nivel 1), el algoritmo se basa en tomar el primer nodo de la base que el usuario definió antes. Posteriormente se hará una intersección mediante planos a todas las caras de la edificación que se encuentran presentes en el nivel 1. Para generar estos planos se necesita de un punto inicial P_0 representado como un punto azul y de un vector perpendicular al plano que se generará. Este vector se obtuvo a partir del siguiente punto que define el usuario, P_1. Para el ejemplo mostrado, el nivel 1 tiene cuatro caras. Entonces el algoritmo recorre cada una de estas, generando un plano e intersecando con el perímetro (4 veces). La intersección de una superficie con una curva da como resultado un punto, en el ejemplo estos puntos son los candidatos a ser el nodo final de la columna. Si observamos ahora la imagen derecha de la figura 3.49, se generan 4 líneas con el nodo inicial azul y distintos nodos finales. En la estructuración de edificaciones es difícil, aunque no imposible, que se propongan columnas con tal inclinación como las denotadas con puntos finales en color rojo. Todo dependerá del diseño arquitectónico y la estética que se le quiera dar. Sin embargo, tomaremos el mayor porcentaje en ejemplos de estructuración de edificaciones y se recomendará tomar la columna más eficiente, la que tiene la mayor verticalidad. Para esto se tuvo que hacer una selección de las líneas generadas, y se tomo como restricción el ángulo con respecto al eje vertical Z. De esta manera el nodo verde del ejemplo, resulta el único que cumple la restricción y por lo tanto es el que se toma para almacenarlo como columna.

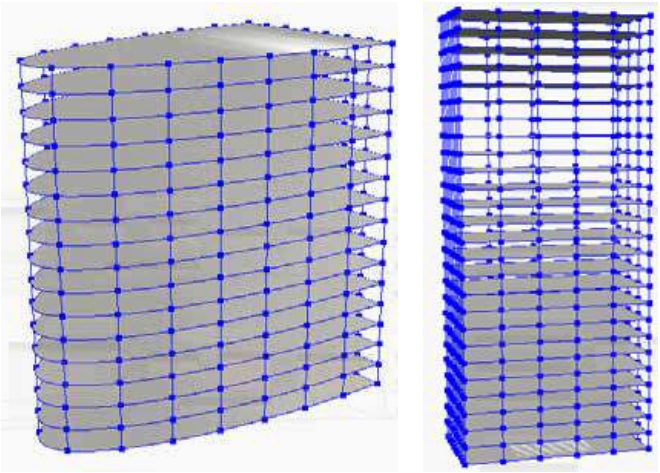


Figura 3.50: Estructuración Externa Estructuración externa de un sólido rectangular mediante el método de las intersecciones en niveles.

Hay que tomar en cuenta que la figura 3.49 solo muestra la primer iteración en el primer nivel, por lo que esto se repetirá por cada nodo que el usuario defina en el inicio y en cada nivel de la estructura. Si el edificio cuenta con muchos niveles, o que en lugar de 4 caras tuvieras más, o que el usuario definiera un número alto de columnas por cara, el número de operaciones se incrementaría mucho. Es por ello que el costo computacional de esta metodología resulto ser muy alto. Sin embargo para geometrías mas o menos regulares, con un número no muy alto en columnas y niveles, el algoritmo permite generar toda la estructura exterior de un sólido sin que el usuario tenga que conocer con exactitud la forma de la edificación. Es decir, de acuerdo a los parámetros iniciales el algoritmo se encarga de identificar la geometría y proponer la posición de las columnas y vigas que la soportaran. Las imagenes de la figura 3.50 muestran algunos casos exitosos de este módulo.

En modelos prismáticos regulares como los mencionados antes, la estructuración en base a intersecciones de niveles funcionó muy bien pero en edificaciones donde las plantas variaran de acuerdo a la altura se tuvieron problemas. Este es el caso del modelo de prueba de la Torre Mayor, donde la sección inicial no tiene problemas al no variar las plantas, pero en la sección superior existe un chaflán en forma de elipse que "corta" la edificación gradualmente mientras la altura crece. Al someterlo al proceso de estructuración se obtuvo el resultado de la imagen 3.51.

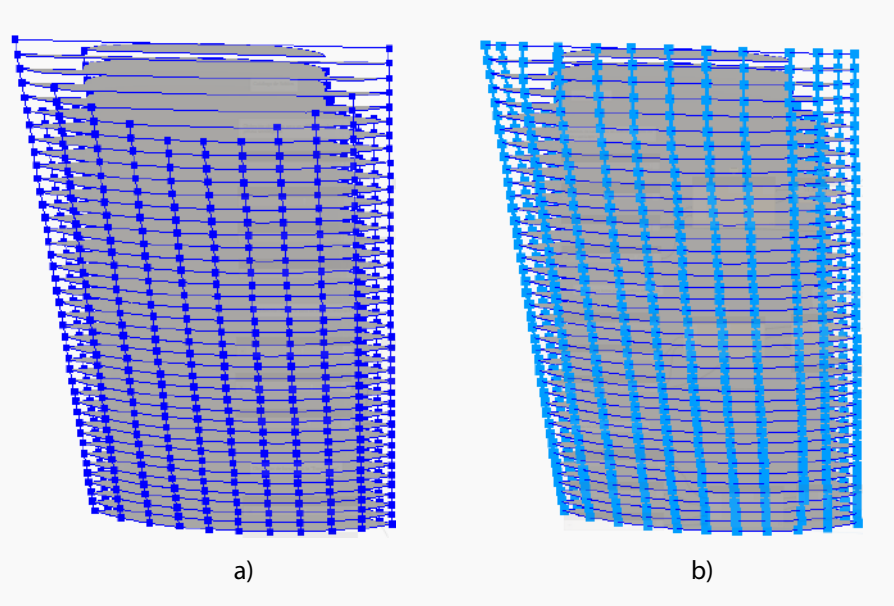


Figura 3.51: Estructuración Externa en Torre Mayor.
a) Se observan problemas debido a que las plantas de la sección superior de la torre se "cortan" por un elipse y por tanto las columnas presentan una inclinación.

b) Se impusieron condiciones de ángulo y longitud para resolver el problema

El problema que observamos en la figura 3.51 a) se debe a la restricción de verticalidad, la cual funciona para seleccionar la columna ideal. La cuestión es que si se desea estructurar un sólido que tenga una característica especial como lo es la Torre Mayor, donde algunas columnas superiores tienen una inclinación, el algoritmo no permitirá generarlas. Una solución que parece trivial es aumentar el ángulo de restricción, es decir "relajar" la condición. Entonces, ¿cuál es el ángulo a utilizar? o ¿qué tanto debemos relajar la restricción para que la selección sea adecuada?, lo que se hizo fue incrementar el ángulo poco a poco. Pero al hacer esto, el problema que surge es que más columnas además de la deseada puede que entren en la condición, entonces el resultado sería que para unos nodos se generarían múltiples columnas que son permitidas por ese ángulo relajado. Por lo que no es tan fácil dar una regla de estructuración para este tipo de sólidos. La solución final fue hacer una combinación de restricciones, se utilizó un "ángulo relajado" mas o menos razonable a utilizar en columnas, además se empleo una restricción de longitud. Si observamos bien las imágenes de estructuración, vemos que las columnas ideales además de ser las que tienen mayor verticalidad también son las que tienen menor longitud. Entonces revisando estas dos condiciones por cada candidato, se obtuvo el siguiente resultado para la Torre Mayor en la imagen 3.51 b).

Aunque los resultados de este planteamiento fueron mejores, aún se tuvieron problemas con la condición de ángulo. Al definir un ángulo que fuera lo suficientemente grande para aceptar las columnas de la fachada inclinada, algunas columnas que debieron ser completamente verticales tuvieron un com-

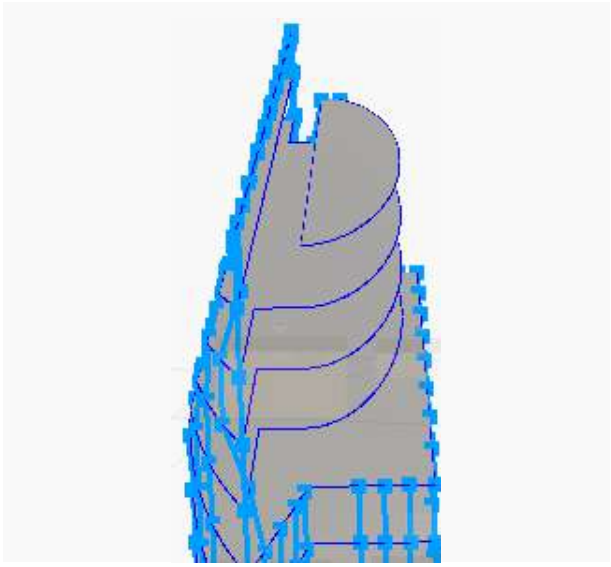


Figura 3.52: Inclinación en columnas. Resultado de estructuración utilizando condiciones de ángulo y longitud probados en el modelo Torre Mayor, con un efecto no muy deseable.

portamiento extraño, en donde presentan una ligera inclinación. Aunque la mayoría de columnas con este problema presentan una inclinación muy pequeña, la sumatoria de estos errores desde el primer nivel hasta el último generan una inclinación no muy deseable como el detalle mostrado en la figura 3.52. Incluso con el hecho de tener este error de inclinación, se pueden generar estructuras interesantes con sólidos que tienen un aspecto estético no tan común. Como el sólido mostrado en la figura 3.53, que posee una forma curva o de tipo "orgánica" por el efecto visual que produce. La estructuración exterior se logra utilizando este algoritmo y se obtiene el resultado de la figura 3.53. Se puede observar que la estructura intenta describir la forma que tiene el sólido para su estructuración.

Los resultados mostrados, aunque son bastante buenos, necesitan "pulirse" haciendo más precisa la selección de columnas mediante las restricciones. Por esa razón se decidió dar una pausa a este módulo para generar otra parte fundamental en la estructuración, la cual se refiere a la generación de todos los elementos estructurales internos.

3.5.3. Estructuración Interna

La lógica de este código es por mucho más simple que el visto en la sección anterior 3.5.2 referente a la estructuración externa mediante intersección de niveles. No es más que una correcta utilización de la información generada, ordenarla de acuerdo a los objetivos deseados y crear nuevos elementos en base a esta.

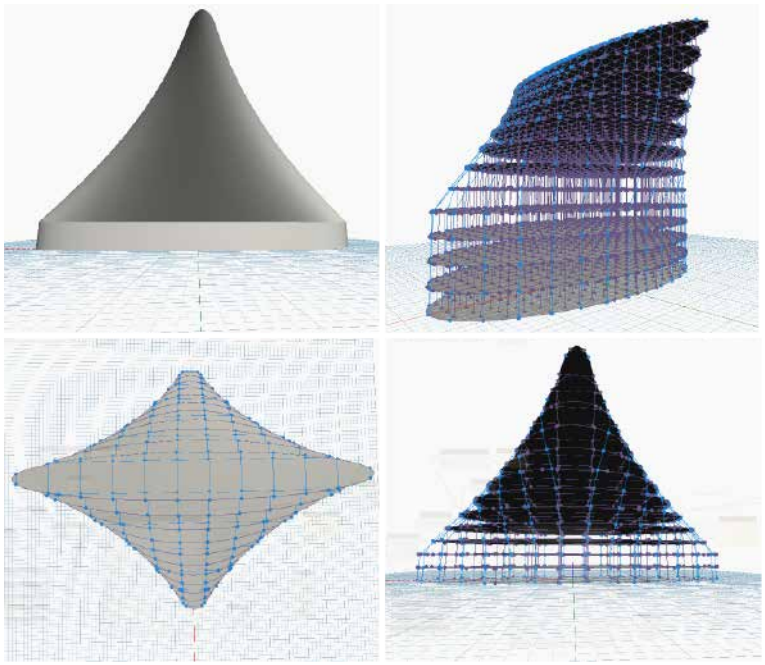


Figura 3.53: Resultado de estructuración externa. Estructuración externa de un sólido con forma orgánica, se muestran algunas vistas laterales y en planta del resultado.

La primer función del código denominada `get_int_cols_nodes(surfaces, n_floors, height, first_f, final_f, curves, grid)` utiliza la información generada por el módulo ***Grid*** y su meta es obtener un arreglo de los nodos que se encuentran dentro de cada superficie de nivel, los cuales servirán para generar las columnas internas. Como se ha mencionado antes, la herramienta de estructuración que el usuario tendrá forma de operar mediante parámetros se basa en una retícula que se sobrepone al sólido tridimensional, esta metodología permite ver donde se ubicarán las columnas mediante puntos (ver sección 3.3). Posterior a que ya tenemos la posición de las columnas se ejecuta la función denominada `get_t_beams(surfaces, height, spa_in, first_f, final_f, curves,grid_nodes_floor,grid)` la cual es la función base para la estructuración interna. A partir de los nodos que se tienen almacenados en un arreglo, se hará uso de un algoritmo de ordenación que los acomodará de cierta manera para generar vigas internas. El planteamiento utilizado en estos nodos es muy similar al que se realizó en los centroides de las losas que se obtuvieron en el módulo ***Grid*** (ver sección 3.3.2). Se hicieron dos copias del arreglo de nodos por cada nivel, primero uno respecto al eje coordenado X, y otro respecto al eje Y. El proceso[1] es el siguiente, primero crear y ordenar sub-vectores de nodos de la manera que tengan la misma coordenada X, de menor a mayor. Después en cada sub-vector con el mismo valor de X, se ordena pero ahora respecto a Y. De esta manera se tiene conocimiento de donde está cada nodo. Al inicio se pensaba erróneamente que al usar funciones de *Python*, éstas serían muy lentas (conociendo que este es un lenguaje de alto nivel, se

[1]Para explicar la metodología se utilizará como ejemplo el arreglo respecto a X.

sabe que su rendimiento es inferior a lenguajes como lo es C, sin embargo no por ello sus funciones son malas), por ello se implementaron algoritmos muy simples de ordenación como lo es el "Bubble Sort", sin embargo tiempo después en la etapa de evaluación de los códigos, se determinó que las funciones como `sorted` o `.sort` propias de Python son altamente eficientes respecto a los algoritmos burbuja, por tal resultado se prefirió utilizar estas funciones para la ordenación de nodos, reduciendo un porcentaje grande de tiempo.

Al tener dos listas de nodos por cada nivel, una ordenada respecto a X y otra respecto a Y, se comienza el proceso de generación de vigas. El proceso es muy simple en realidad, primero mediante un ciclo se recorre la lista de nodos ordenados con respecto a X, si estos están contenidos en el mismo subvector, es decir que tienen la misma coordenada X, se unen mediante líneas que representan idealizaciones de vigas. El mismo procedimiento se realiza con la lista de nodos ordenados con respecto a Y. De esta forma ya conseguimos una retícula formada por líneas o idealizaciones de vigas que se utilizarán en el análisis estructural. Hasta este punto ya teníamos un algoritmo que generaba estructuración externa (ver sección 3.5.2) y ahora junto con el código descrito antes ya podemos generar las líneas que representan la estructuración interior en base a una retícula. Algunas imágenes que denotan esta implementación se presentan a continuación.

Cómo podemos ver en la figura 3.54, la estructuración externa e interna del modelo "Torre Mayor" se ve bastante bien. Parece tener lógica, por lo que podría ser funcionalmente correcto proponer una configuración como la mostrada. Sin embargo, los dos algoritmos que generan los elementos estructurales están aislados, cada uno realiza su función sin interactuar con el otro, por lo tanto tenemos dos sistemas estructurales por separado. Esto en un análisis estructural no sería muy recomendable, ya que no existiría un comportamiento en conjunto, sino que cada estructura trabajaría por su cuenta. El objetivo es generar una sola estructura (externa e interna) que su comportamiento sea en conjunto y además que sea posible de analizar mediante el análisis matricial de las rigideces. Para ello se pensó en varias soluciones:

1. Fusionar los dos algoritmos (estructuración externa e interna).
2. Generar la estructuración interna en base al algoritmo de estructuración externa.
3. Generar la estructuración externa en base al algoritmo de estructuración interna.

La opción 1 sobre fusionar ambos algoritmos trataría de ejecutar ambos procesos y de alguna manera conectar los elementos estructurales de acuerdo a como se propusieron por el usuario. Esta idea es complicada desde el punto de

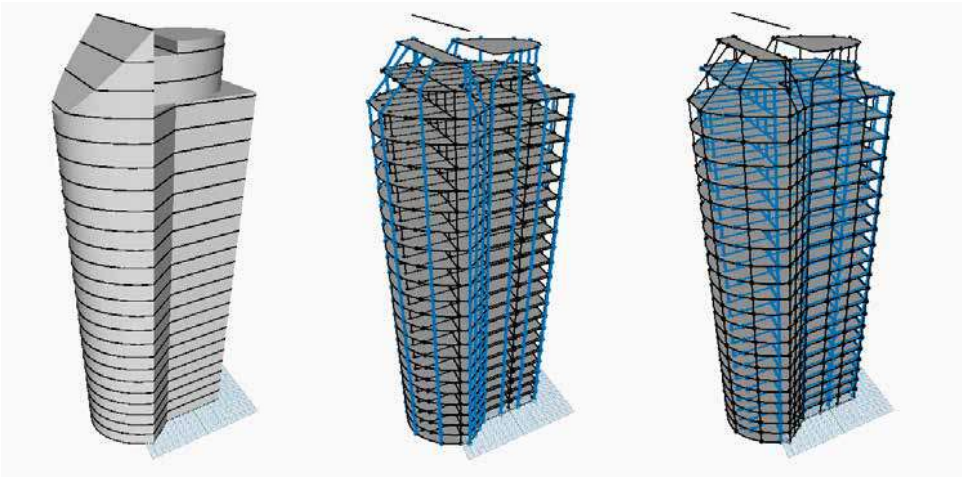


Figura 3.54: Resultado de estructuración externa e interna. Estructuración externa e interna del modelo experimental "Torre Mayor", se observa con color azul la separación de la estructuración por cada módulo.

vista que además de graficar líneas que representen la estructura debe existir una topología congruente para que el análisis se realice con éxito. Por lo tanto al tener que buscar las intersecciones de líneas entre uno y otro algoritmo, dar nuevas conectividades a éstas e inclusive evitar que existan líneas sobrepuestas no es una tarea sencilla. Aunque no es imposible de resolver este problema se dejará para un trabajo futuro. La opción 2 y 3 son conceptos similares, aunque se debe buscar la opción indicada de acuerdo a la utilidad de la aplicación. Por lo tanto se determinó que generar una estructura interna en base a lo que el usuario propone para la estructura externa es un trabajo tan complicado o aún mayor que la opción 1, ya que como recordaremos en lo visto en la sección 3.5.2 la forma de proponer una configuración es mediante número de separaciones por cada curva, entonces si pensamos en una edificación como la Torre Mayor que tiene múltiples caras distintas, la conexión de vigas interiores se dificulta y el resultado quizá no sería funcional, ya que habría zonas donde se deberán unir elementos estructurales paralelos o incluso eliminarlos por que no tienen el mismo número de conexiones en las cara encontradas. Las problemáticas anteriores sugieren que la opción 3 basada en la retícula uniforme es la que más se adecua al tipo de proyecto. Esto tiene sentido, ya que al definir separaciones regulares entre columnas se pueden generar los elementos internos, además los elementos externos se pueden obtener de la misma manera, solamente modificando un poco las opciones mencionadas en esta sección. En cuanto al algoritmo de estructuración externa mediante intersecciones en niveles, se dejará en fase de experimentación por el momento, y permitiendo que se retome como trabajo a futuro.

3.5.4. Estructuración Reticular Externa e Interna

Realizar una estructuración en base a una retícula tiene ciertas ventajas de programación y utilidad, las cuales se comentaron en la sección anterior. Sabemos que la mayoría de las edificaciones se proponen con columnas que tienen separaciones regulares, al igual que las vigas. Esto denota una estructuración en base a retícula y permite un análisis y diseño más eficiente debido a que se busca tener simetría en cuanto a las acciones mecánicas. A diferencia de algunas edificaciones que al buscar mayor estética necesita de planteamientos complicados que rompen con el esquema tradicional de estructuración, se dejaran de lado en este trabajo debido a que se busca que un mayor número de edificaciones puedan ser estructuradas con este código.

Ahora para este planteamiento se añaden algunas funciones en `get_t_beams (surfaces, height, spa_in, first_f, final_f, curves,grid_nodes_floor,grid)`. El código de estructuración utilizado se muestra en el apéndice A.6. Lo primero que cambia con respecto a la versión de estructuración interna es que a la hora de agregar los nodos al arreglo mencionado antes, también se añadirán los nodos iniciales y/o finales de curvas externas, para tener mayor información a la hora de generar las vigas exteriores. También se añadió una función denominada `ext_beams(curve,ext_nodes)` la cual tiene por objetivo generar las vigas exteriores de la planta. La forma de obtener líneas externas no es tan sencilla debido a que no basta tener los puntos extremos, porque al no tenerlos ordenados de cierta forma, el resultado sería un desastre de líneas que no tienen sentido, éstas estarían cruzando la planta y no serían útiles, tal como lo muestra la figura 3.56. En la *figura a)* se puede observar que las figuras que componen al modelo experimental *Torre Mayor* tienen cierta dirección que asigna **Dynamo** desde que se crearon dichas funciones, el problema de generar sólidos compuestos por estas figuras, es que a la hora de hacer las intersecciones, las direcciones de cada cara se siguen conservando. Esto lo muestra la *figura b)* ya que el conjunto de líneas no siguen la misma dirección como lo muestran las flechas en color rojo. Aunado a esto, al utilizar el módulo ***Grid*** para proponer la estructuración (como es denotado por líneas verde-azul), se hacen intersecciones de líneas para obtener puntos extra. Entonces tenemos un arreglo de puntos que el sólido proporcionaba (índices color negro) y tenemos ahora puntos añadidos por la estructuración del usuario (índices color rojo). Este es un problema porque al ejecutar la función de estructuración externa, las líneas no tendrán lógica estructural como se ve en la *figura c)*.

Figura 3.55: Problemática con vigas externas . Se muestra el problema de la generación de líneas externas en el modelo experimental *Torre Mayor*.

Fig. a) Dirección de figuras básicas de **Dynamo**.

Fig. b) Se añaden puntos (índice rojo) debido a la propuesta de estructuración.

Fig. c) La estructuración de esos puntos siguiendo el orden de almacenamiento regresa líneas sin lógica estructural.

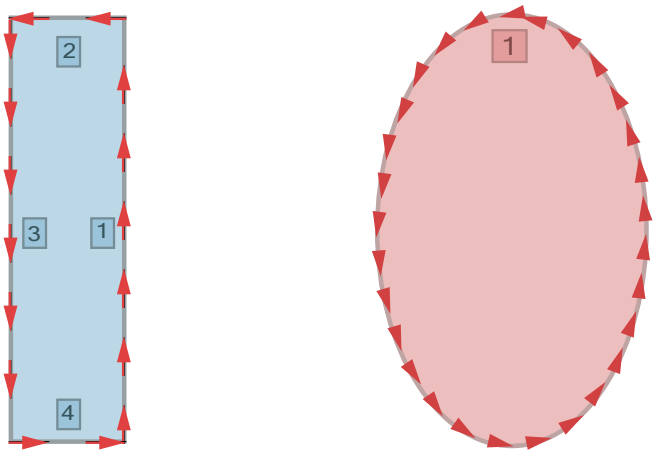


Fig. a)

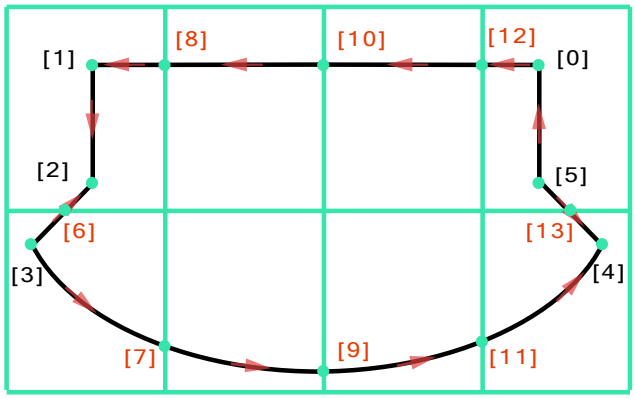


Fig. b)

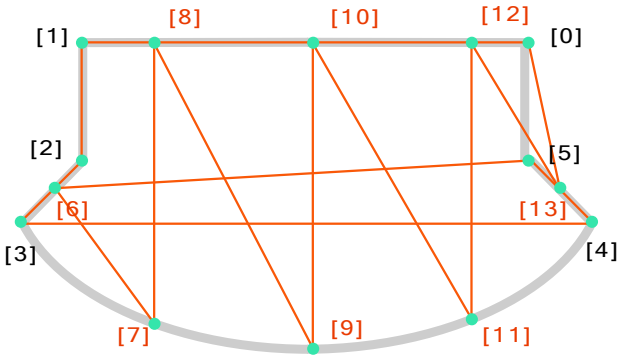


Fig. c)

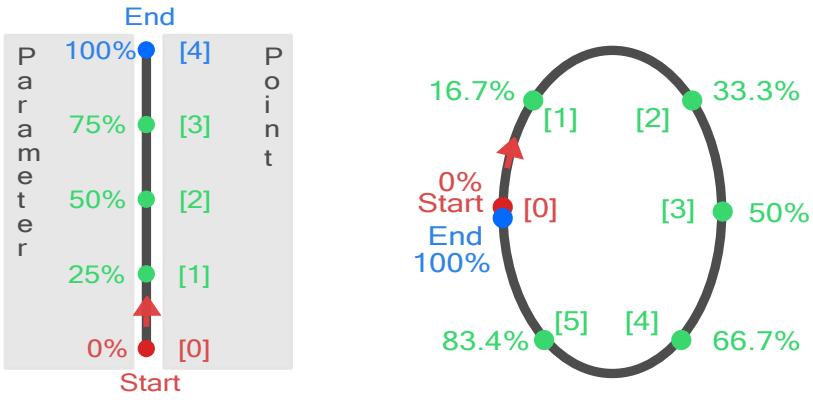


Figura 3.56: ParameterAtPoint. Esta imagen muestra el funcionamiento *"Bidireccional"* de la función *Curve.ParameterAtPoint(Points[])* y su inversa *Curve.PointAtParameter(Parameters[])*, las cuales son mostradas de forma gráfica con dos ejemplos, una línea recta y una elipse.

La forma de resolver este problema fue mediante una estrategia que utiliza una función propia de **Dynamo** denominada `Curve.ParameterAtPoint(Points[])` y su inversa `Curve.PointAtParameter(Parameters[])`. Lo que hace la primer función es que los puntos que estén intersecando a una curva se les dará un número del 0 al 1 (porcentaje) de acuerdo a su posición del inicio de la curva al final de ésta. En curvas perimetrales como las que se obtienen en el módulo ***Floors*** revisado en la sección A.2, es posible utilizar esta metodología aún cuando son curvas cerradas, ya que también poseen un inicio y un final, por ello obtener un porcentaje de acuerdo a su posición en la curva es posible. Como se muestra en la figura 3.56, la función tiene dos formas de utilizarse, es decir, al realizar un proceso es posible obtener la función inversa de la información. Ahora, tome en cuenta la recta mostrada, suponga que divide dicha recta mediante puntos, si utiliza la función `Curve.ParameterAtPoint(Points[])` se obtendrá un vector con el porcentaje de corte en cada punto. De manera inversa, si se tiene un vector con datos porcentuales como por ejemplo $[0, 0.25, 0.5, 0.75, 1]$, al emplear la función `Curve.PointAtParameter(Parameters[])` se regresarían objetos punto con coordenadas geométricas definidas por los parámetros y la recta. Si se desea realizar este procedimiento en un polígono, un círculo o cualquier geometría cerrada, la función hará su trabajo adecuadamente.[1] Ya que **Dynamo** asigna un punto de comienzo y coincide con el punto final tal como se muestra en 3.56, debido a que tiene asignada una dirección por defecto y de la misma manera se obtienen los *parámetros*.[2]

[1] Se han tenido problemas aislados en donde la función tiene cierta inestabilidad en objetos geométricos como curvas. Esto ha sido investigado a detalle e incluso desarrolladores de la comunidad **Dynamo** no tienen respuestas, por ello se dejará como trabajo futuro para dar mayor flexibilidad al código.

[2] La palabra *Parámetros* dentro del contexto de las funciones `Curve.ParameterAtPoint(Points[])` y su inversa, tiene un significado distinto al utilizado antes. Aquí se refiere a un valor numérico porcentual que indica la posición de intersección de un punto sobre una curva.

Tener el porcentaje de corte por cada punto externo en una curva nos sirve para conocer su posición y ordenarlo de acuerdo algún objetivo, en este caso se requiere para la obtención de vigas externas. Revisando la figura 3.56 c), se observó que debido a la metodología de estructuración, se obtienen puntos en el perímetro de forma desordenada, entonces utilizando las funciones `Curve.ParameterAtPoint(Points[])` y su inversa `Curve.PointAtParameter (Parameters[])`, se logrará ordenar los puntos para ahora sí generar las vigas externas. La forma de implementarlo se puede observar en la figura 3.57. La figura a) es la planta del modelo experimental *Torre Mayor* ya conocido, donde se tienen los puntos debido al sólido (negros) y los puntos de estructuración (rojos), además de la posible dirección de cada línea. En la caja color gris se muestra el vector de puntos (representados por sus índices de posición en el vector), al ser sometidos a la función `Curve.ParameterAtPoint(Points[])`, se obtiene un vector de igual tamaño con el porcentaje de corte en el perímetro de la superficie. Tal cual no tienen un uso, a menos que se aplique el algoritmo de ordenación sobre estos *parámetros* como se observa en la figura b) con la función *"sort"*, de esta manera tenemos en la caja verde los mismos porcentajes ordenados de menor a mayor, es decir que desde el inicio de la curva hasta el final. Aplicando ahora la inversa, `Curve.PointAtParameter(Parameters[])`, se obtiene una lista de puntos (representados en índices color anaranjado). Estos puntos ya tienen las coordenadas geométricas mostradas en la figura c), por lo que al unir los puntos se obtienen las vigas externas.

Con los puntos externos e internos, se hace una comprobación de que todos se encuentran dentro de la superficie que delimita la losa de cada nivel. Dicha verificación se hace mediante una intersección de los nodos con la superficie. La problemática de utilizar la función `Geometry.Intersect(Geometry)` de **Dynamo** es que algunas veces regresa listas vacías o elementos que no se tenían contemplados. Por ejemplo una intersección entre nodos y una superficie debería devolver nodos, pero algunas veces regresa curvas. La forma de eliminar elementos no deseados o "basura" de las listas es mediante una función que se programó denominada `list_filter(list, filter_string)`. Esta función recorre cada elemento de la lista indexada, y usando la función `element.GetType()` se verifica si es del tipo especificado en la variable `filter_string`. De esta manera se pueden limpiar las listas después de una intersección lo cual es muy recomendable cada vez que se haga esta operación.

Posteriormente se ejecuta la operación que se describe en la sección 3.5.3. Mediante la ordenación de nodos (ahora incluyendo los externos) respecto a X y respecto a Y, y posteriormente generando las vigas interiores. Ahora solamente hay que verificar que no se repitan vigas ya que en modelos donde la retícula base esté empalmada con una cara del sólido como podría pasar en prismas rectangulares. Esto se hace comparando cada viga externa con las vigas internas recién regeneradas. Al terminar este proceso ya se tendrán vi-

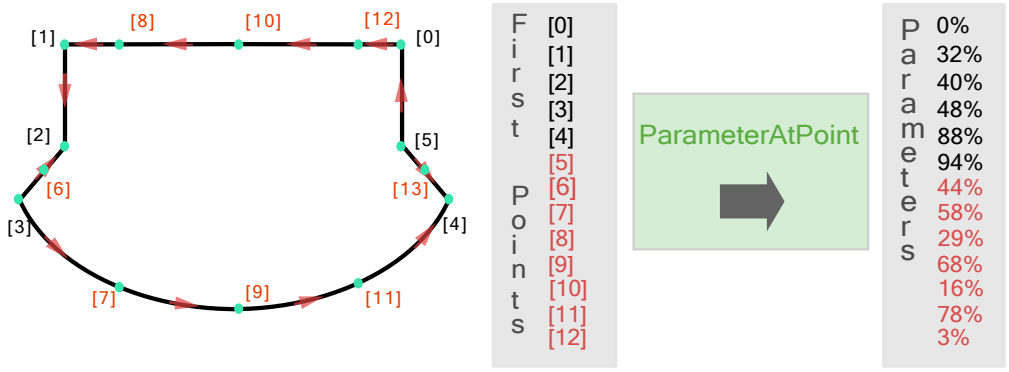


Fig. a)

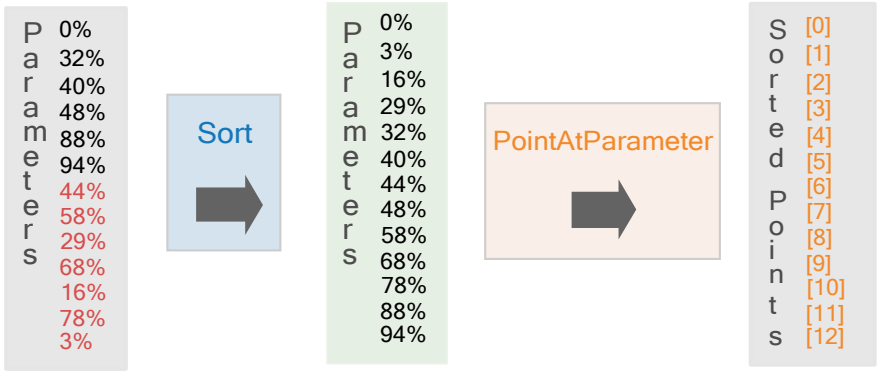


Fig. b)

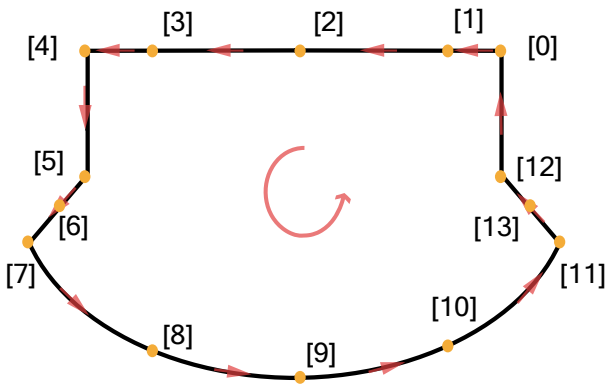


Fig. c)

Figura 3.57: Ordenación de Puntos. Se observa la apliación de las funciones `Curve.ParameterAtPoint(Points[])` y `Curve.PointAtParameter(Parameters[])`, en una planta de la *Torre Mayor*.

sualizadas las líneas de todos los elementos estructurales que necesitábamos y en un mismo conjunto. No obstante, aún no tenemos la información que necesitaría un análisis estructural matricial debido a que no hemos etiquetado a las barras ni a los nodos, además esto significa que tampoco le hemos dado orientación a las barras porque no definimos la topología estructural.

Para adjuntar esta información a los elementos estructurales se hizo uso de el paradigma que ofrece el lenguaje *Python*, referente a la programación en base a objetos. Se creo el objeto `frame`, el cual contiene información relevante como la numeración de la barra, el nodo inicial y el nodo final. Para asignar la numeración a la barra simplemente se crea un objeto `frame` por cada barra en la estructura mediante un ciclo que recorre la lista de barras, se agrega la línea con la función de clase `frame.add_frame(element)` y el índice se guarda en un entero dado por el ciclo. La numeración de nodos se añade de la misma manera al crear un objeto `node` y guardar su índice mediante un ciclo. Un ejemplo de clase *frame*, se muestra a continuación.

```
1
2  class frame:
3
4    def __init__(self, id):
5      self.id=id
6      self.loads_vec=[]
7
8    def add_frame(self,added_frame):
9      self.frame=added_frame
10
11   def add_ini_point(self, added_ini_point):
12     self.ini_point=added_ini_point
13
14   def add_fin_point(self, added_fin_point):
15     self.fin_point=added_fin_point
16
17   def __del__(self):
18     print #Destructor
19 .
```

Listing 3.4: Clase *frame*.

Con la numeración de nodos y barras solo restaría un paso para tener la información estructural que necesita cada barra, la orientación. Ésta se obtiene definiendo el nodo inicial y el nodo final de cada barra, para ello se programó una función denominada `search_nodes_frames(frames_index,frames,nodes_index,nodes)` . Esta función quizá es una de las que ocupan más porcentaje de tiempo en todo el módulo de estructuración, ya que su objetivo es realizar comparaciones de nodos con barras para conocer si están conectados entre sí, además de saber cual es el nodo inicial y el final. Mediante ciclos anidados, la función utiliza las coordenadas geométricas de los nodos ya ge-

nerados para comparar con los nodos iniciales mediante la función de **Dynamo** `Curve.StartPoint()`. Si ésta da como positivo, una bandera da la indicación de que el nodo es inicial, de la misma manera utilizando la función `Curve.EndPoint()` se realizá una comparación para saber si el nodo era final. Al final se devuelve un arreglo de listas que contiene el nodo inicial como primer elemento y el nodo final como segundo, el arreglo se ordena de acuerdo a la numeración de cada barra. Esta función hace comparaciones de todos los nodos y todas las barras que están en un mismo nivel, por lo que tiene un alto costo computacional, se recomienda para un trabajo futuro intentar optimizar esta parte reduciendo el campo de búsqueda. A pesar de eso, de esta manera ya tenemos la orientación de cada viga en la estructura de la edificación.

Al terminar la función de obtención de vigas, solamente restaría generar las columnas para tener completo nuestro modelo estructural. La función encargada de esto se denominó `get_int_cols(first_f, final_f, nodes)` y el procedimiento que realiza es muy sencillo en realidad. Debido a que ya tenemos los nodos por cada nivel donde las vigas se conectan, ahora solo restaría conectar líneas en sentido vertical para formar las columnas. La forma de hacer esto es recorriendo el vector de nodos, del piso inferior con el piso superior, y comparando su posición mediante sus coordenadas. Al terminar este ciclo se le asigna un índice a cada columna y se ejecuta la función `search_nodes_frames(frames_index,frames,nodes_index,nodes)` que se explicó antes para conocer las conectividades de cada barra.

La siguiente función que se ejecuta se denominó `material(beams_mat, columns_mat, beam_index, column_index, first_f, final_f)` y para conocer su funcionamiento es necesario revisar la sección 3.4 referente al módulo de *Propiedades de los Materiales y Secciones Transversales* ya que ahí se explica cómo se obtiene la información que hasta este punto se utilizará. Como se ha mencionado, ésta es una herramienta en fase de planeación y además considerando que en edificaciones reales no se varía mucho las secciones transversales para estandarizar el proyecto, las secciones y el material empleado solo podrán variar por nivel. Es decir que no habrá distintas secciones de columnas o vigas en un mismo nivel. Dada esa consideración, el primer paso dentro de la función `material()` es almacenar los tipos de secciones en un vector para vigas y en otro para columnas. Posteriormente mediante ciclos anidados se verifica el material empleado por cada nivel y se recorre el vector de vigas, si éstas se encuentran en el mismo nivel de cierto material, se agrega esta información a un vector. De la misma manera se realiza este procedimiento para las columnas. Al final se tendrán vectores con el número de material propuesto de acuerdo a los catálogos que el software **MECA** maneja para su análisis y verificación en el diseño.

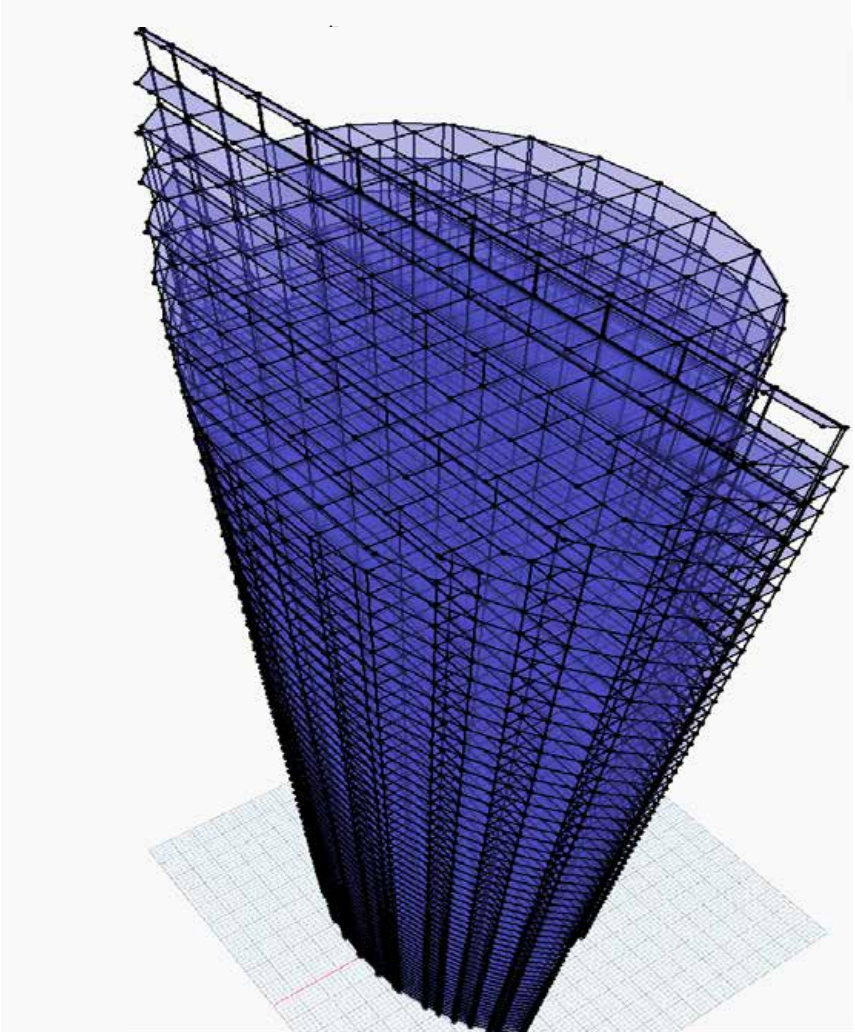


Figura 3.58: Resultado de Módulo *Structure*. Se muestra el modelo estructural resultante de la Torre Mayor, el cual tiene conectados todos los elementos estructurales para que su análisis pueda mostrar el comportamiento en conjunto.

La última parte del módulo reúne la información geométrica y topológica necesaria para el análisis estructural. Por lo que ***Structure*** retorna los parámetros obtenidos por módulos anteriores y además un vector que contiene información de columnas, vigas y materiales seleccionados. Ahora lo que faltaría para realizar el análisis es asignar fuerzas a las que una edificación es sometida.

3.6. Fuerzas Involucradas en Edificaciones

Un modelo estructural puede estar sometido a muchos tipos de solicitaciones o fuerzas de muy diferente índole. Las principales fuerzas que se estudiaron en el desarrollo del análisis estructural son las causadas por el efecto de la gravedad sobre los cuerpos, por tal motivo se denominan fuerzas gravitacionales. Dichas fuerzas se deben a los elementos que está soportando la estructura, es decir, al peso de la edificación o de quienes habitan dicha estructura. La carga causada por las personas que habitan un espacio o los elementos que son colocados sobre la estructura debido a su función como por ejemplo muebles de oficina, estantes para bibliotecas, entre otros, se denominan en conjunto como *Carga Viva*. Mientras que el peso propio de los elementos estructurales, que también aportan fuerzas considerablemente grandes, son clasificados en otro tipo de conjunto de fuerzas denominado *Carga Muerta*. Aunque el valor de estas fuerzas se considera tal cual se calcula, la verificación de resistencia por diseño regularmente pide que estas cargas se les agregue un factor para incrementar su valor y así tener cierta seguridad para acciones involuntarias.

3.6.1. Metodología de Distribución de Fuerzas en Losas

Las cargas gravitacionales son muy importantes en el análisis de edificaciones. Por tal motivo el módulo de asignación de fuerzas solo consideró estas acciones como primer etapa de proyecto, sin embargo se propone como trabajo a futuro generar más módulos de asignación de fuerzas como podrían ser causadas por viento o sismo. La metodología empleada en la asignación de fuerzas se basa en un procedimiento geométrico, el cual es usado por la mayoría de oficinas de Ingeniería estructural e incluso es usado por software comercial. La idea en general es utilizar magnitudes de carga por unidad de área que varían de acuerdo al tipo de edificación que se diseñará. Por cada losa propuesta en el modelo se debe distribuir cierta cantidad de fuerza, mediante áreas denominadas *tributarias*, a las vigas que soportan dicha losa. No se encontraron fuentes bibliográficas que demuestren que esta distribución mediante áreas es precisa, sin embargo se encontraron manuales de software y referencias digitales que promueven este método. Además durante las clases en Ingeniería Civil, se enseñó que la forma de asignar las cargas de losas a vigas es utilizando este método. Por tales motivos se eligió esta forma de asignar las fuerzas a las barras, además de que **Dynamo** es un modelador geométrico, por lo tanto fue adecuado utilizar un método basado en geometría.

El método geométrico de distribución de cargas depende tanto de la forma de la losa, del número de vigas que soportan la losa, así como del método constructivo a utilizar. En cuanto a la forma de la losa, consideramos que hay tres tipos distintos principales que influyen en la distribución, aunque algunas referencias consideran algunos más.

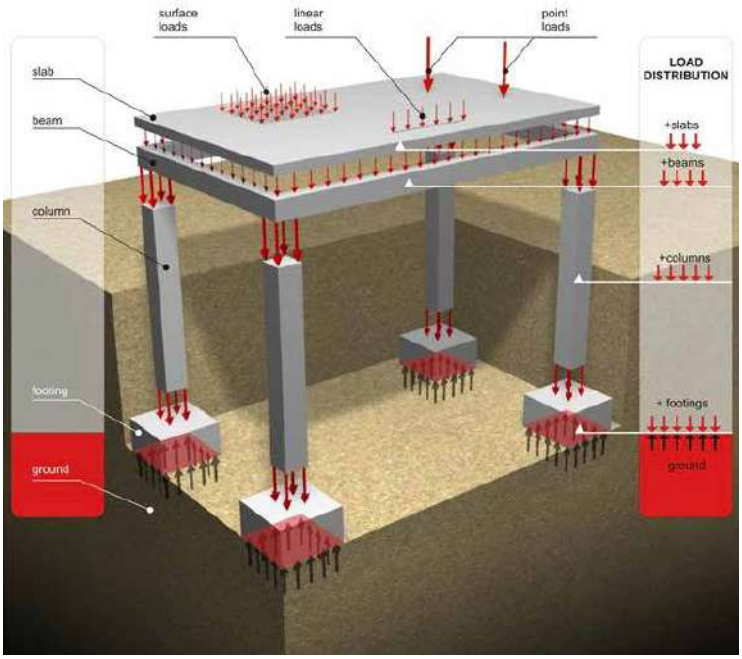


Figura 3.59: Distribución de Fuerzas. Se muestra la distribución de fuerzas que sufre una estructura. Al momento de implementar cargas gravitacionales como las mostradas, los elementos estructurales transmiten las fuerzas comenzando en el punto de aplicación, que por lo regular es en las losas donde se colocan, posteriormente se transmiten a los vigas que sostienen la losa. Las vigas están sostenidas por columnas las cuales transmiten las cargas a la cimentación y ésta es la que distribuye los esfuerzos al suelo (25).

Tipos de Losas:

1. Losas rectangulares o cuadradas.
2. Losas poligonales.
3. Losas cuyo centroide se encuentra fuera de éstas.

En cuanto al inciso 1, tenemos que el método geométrico se basa en trazar triángulos y trapecios según se distribuya a la viga corta o la viga larga respectivamente. El procedimiento en dibujo comienza trazando líneas a 45º a partir de los vértices de cada lado, y encontrando la intersección de dichas líneas, tal como se muestra en la figura 3.60. Para un caso más general, en lugar de usar 45º, se utiliza la mitad del ángulo entre las dos caras. Este procedimiento es aceptado si la losa se encuentra perimetralmente apoyada por vigas. Al tener definidas las áreas tributarias, el siguiente paso es asignar cargas distribuidas a las vigas perimetrales. Para las vigas que les corresponde un trapecio, las cargas se asignarían de la siguiente manera mostrada en la figura 3.61.

Sin embargo el software de análisis **MECA** no tiene implementada la asignación de cargas trapezoidales, o en su defecto de cargas triangulares para formar el trapecio. Por tal motivo se usará el método simplificado, el cual consiste en utilizar la multiplicación de la carga impuesta por el área tributaria, asignando una carga uniformemente distribuida a la viga al dividir entre la carga entre la longitud de ésta, como lo muestra la figura 3.62.

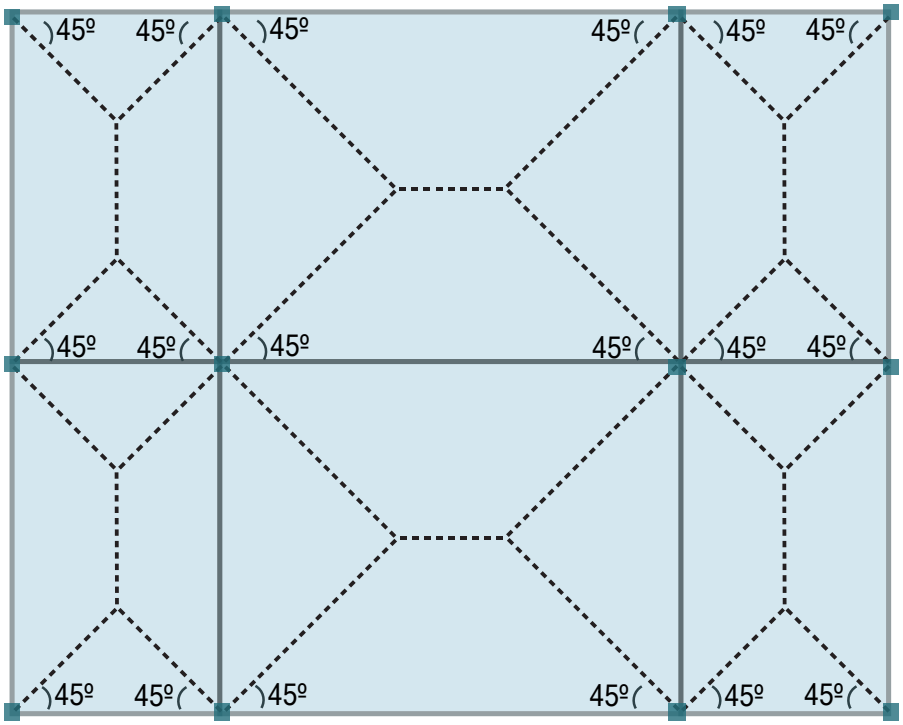


Figura 3.60: Métodos de Distribución de Cargas. Losas rectangulares las cuales muestran la distribución mediante triángulos y trapecios obtenidos de la intersección de líneas a 45º

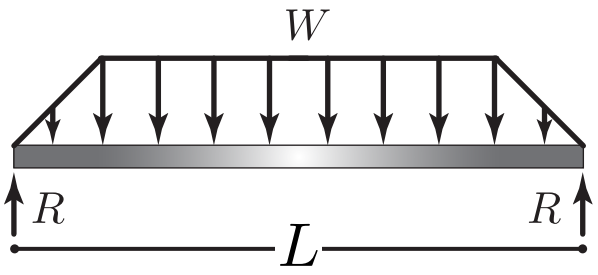


Figura 3.61: Distribución en vigas. Carga repartida en una viga con área tributaria trapezoidal.

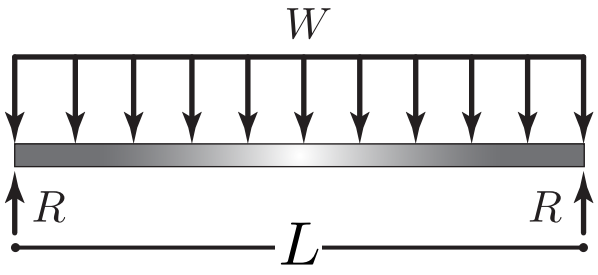


Figura 3.62: Distribución en vigas. Carga repartida en una viga, utilizando el planteamiento simplificado con área tributaria uniforme.

Cómo podemos ver en las imágenes 3.61 y 3.62, la simplificación no repercute tanto debido a que se equilibra la fuerza que aumenta gradualmente desde cero hasta la altura de la carga trapezoidal con respecto a la carga rectangular. Las variaciones más grandes que podremos encontrar es en los momentos en extremo, aunque no es una diferencia tan considerablemente grande como para preocuparse. En los extremos cortos las áreas tributarias son triángulos por tanto las cargas distribuidas en vigas también pueden asignarse como tales, esto se muestra la figura 3.63, pero utilizando el mismo método simplificado, las cargas se asignaron como uniformemente distribuidas.

En el caso de losas cuadradas (fig 3.64), utilizando el mismo método tendríamos que las líneas a 45º convergen en el mismo punto, el centroide. Por ese motivo las cargas se distribuirían triangulares como la figura 3.63, pero de igual manera se simplificará con cargas uniformemente distribuidas.

En cuanto al punto 2, las losas poligonales apoyadas perimetralmente tienen una distribución distinta. Las áreas tributarias dependen del número de vigas y la forma que tiene la losa en general ya que pueden ser polígonos regulares e irregulares. El procedimiento es obtener el centroide de la losa, las áreas son en base a triángulos formados por dos puntos de la viga perimetral y un punto del centroide, como se muestra en la figura 3.65

El último tipo de losa que se clasifica por su forma es el que presenta una disposición tal que su centroide geométrico se encuentra fuera de la misma losa. Esto pasa regularmente con pasillos en "L" o figuras con lados más largo con respecto a otros. Puede ocurrir tanto en figuras poligonales o en figuras rectangulares. La figura 3.66 muestra este caso. La forma de distribuir las cargas de este caso es agregando vigas ficticias embebidas en la losa, que ayuden a continuar con el método geométrico y de esta manera distribuir las cargas. La carga que se reparta en estas vigas ficticias se agregarán como fuerzas puntuales en las vigas reales, esto se muestra en la figura 3.67. En este trabajo se omitió la implementación de este tipo de losas, sin embargo es recomendable que para un trabajo posterior se involucre.

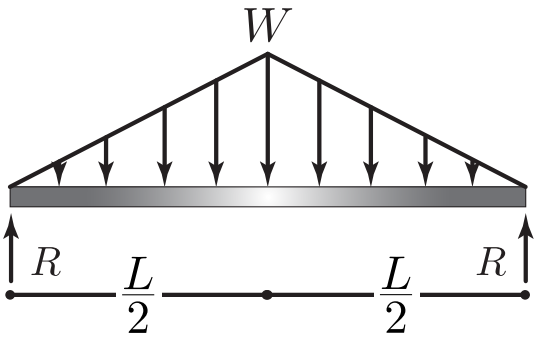


Figura 3.63: Distribución en vigas. Carga triangular repartida en una viga.

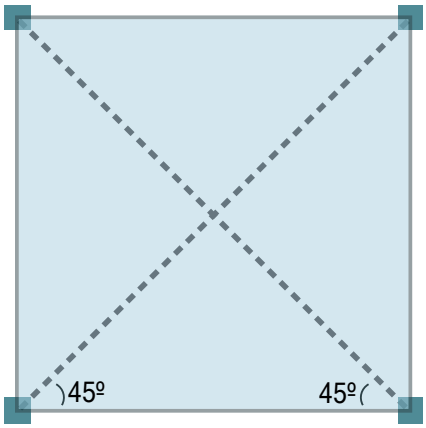


Figura 3.64: Métodos de Distribución de Cargas. Losa cuadrada, las líneas convergen en el centroide de la losa.

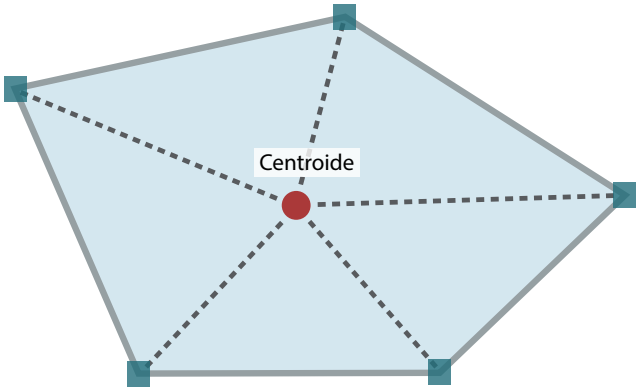


Figura 3.65: Métodos de Distribución de Cargas. Losa poligonal, las líneas forman areas tributarias triangulares y convergen en el centroide de la losa.

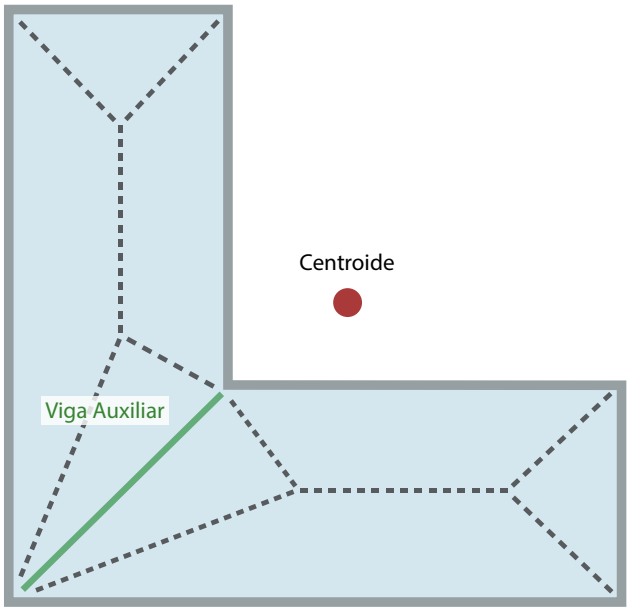


Figura 3.66: Métodos de Distribución de Cargas. Union de dos losas rectangulares en "L", por lo que su centroide se encuentra fuera la misma.

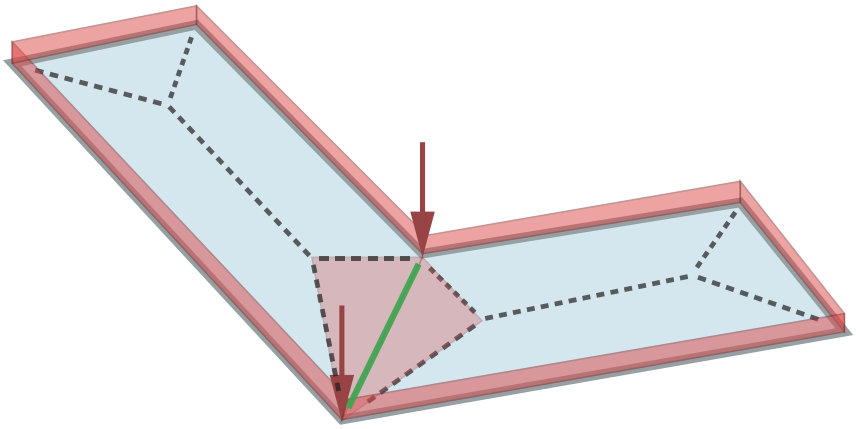


Figura 3.67: Métodos de Distribución de Cargas. Consideración de vigas auxiliares ficticias para distribución de cargas.

Ahora, también la distribución de cargas puede variar de acuerdo al sistema constructivo a utilizar. Si por ejemplo tenemos losas rectangulares pero solamente las apoyamos en dos vigas en lugar de apoyarlas perimetralmente, las áreas tributarias difieren. Este caso también se presenta cuando se proponen losas tipo "Losacero", donde la distribución en lugar de ser *"Bidireccional"* como lo revisamos antes, se considera *"Unidireccional"* como la figura 3.68 lo muestra. La distribución unidireccional no se consideró en este trabajo, se dio prioridad a la bidireccional. Aunque el nivel de complejidad del tipo unidireccional no es tan grande, se consideró dejarlo para un trabajo posterior donde se involucren tipos de sistemas constructivos.

En el caso particular de edificaciones que usen muros de carga de mampostería, la distribución de cargas es de tipo empírica de acuerdo al criterio del ingeniero. Por tal motivo, programar esta metodología es más complicado, se deben tener en cuenta factores constructivos como la ubicación de ventanas y puertas, la figura 3.69 muestra un ejemplo de este procedimiento. Este trabajo no considera esa metodología porque la base de la configuración estructural a analizar es mediante elementos tipo marcos, no con muros de carga.

Tomando las consideraciones que se revisaron antes sobre los tipos de losas y su distribución mediante métodos geométricos, se decidió utilizar solamente la distribución bidireccional de losas perimetralmente apoyada. Por lo que se programaron funciones tanto para losas rectangulares o de cuatro lados que usan trapecios y triángulos, así como poligonales utilizando el centroide. La razón de empezar con este tipo de losas es que el tipo de estructuración que se genera por el módulo ***Structure*** es de tipo marcos o pórticos mediante vigas y columnas, por lo que no se utilizan métodos que involucran muros. Además no se consideró la distribución unidireccional por el momento porque se planea en un proyecto futuro añadir un módulo que permita seleccionar métodos constructivos, por ahora solo se revisarán losas macizas y su distribución. La siguiente sección explica la implementación del método en el código.

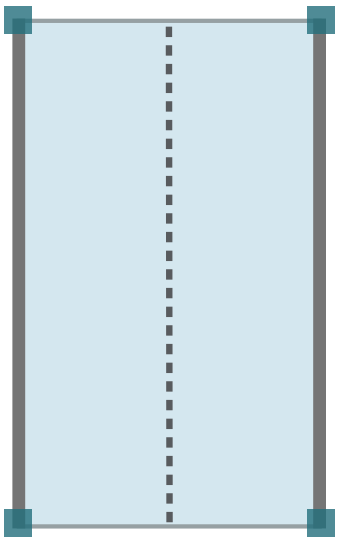


Figura 3.68: Métodos de Distribución de Cargas. Se muestra un ejemplo para losas que requieren una distribución unidireccional de cargas.

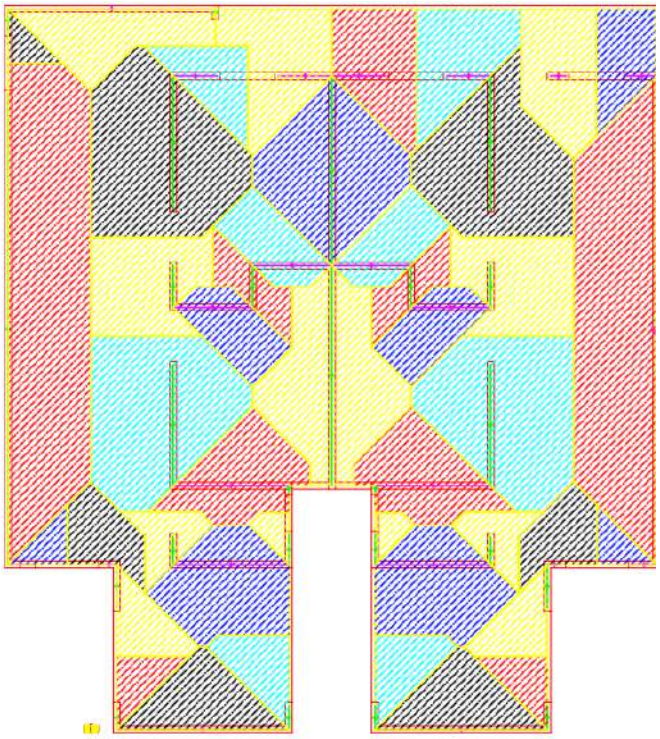


Figura 3.69: Métodos de Distribución de Cargas. Mediante colores se muestra la distribución de cargas en edificación con muros de mampostería, se puede observar la complejidad que pudiera tener la programación de esta tarea.

3.6.2. Implementación en Código de la Distribución de Fuerzas en Losas

El algoritmo de distribución de fuerzas en losas se implementó en un módulo denominado ***Slab Loads***, el cual está contenido en el apéndice A.7. El funcionamiento de éste comienza recibiendo la lista que incluyen la magnitud de cada fuerza superficial que el usuario desee asignar, estas pueden ser por carga muerta, por carga viva, o por cualquier tipo de fuerza superficial que el ingeniero proponga, ya que no tiene un límite de inclusión de cargas. Se pensó que el estándar más utilizado es con unidades de Kg/m^2, por ello al recibir la lista con las magnitudes se hace una transformación a Kg/cm^2, debido a que vamos a utilizar el software **MECA** con unidades de Kg para fuerzas y cm para longitudes.

Posteriormente y dentro de un ciclo que recorre cada nivel se ejecuta la primer función denominada `get_tribu_areas(surf_vec,centroids,beam_points)`. Esta función recorre cada superficie generada por los módulos anteriores y será la encargada de identificar el tipo de losa que es de acuerdo a la clasificación que fue descrita en la sección 3.6.1 de este capítulo. También es importante hacer notar que para este módulo se aprovecharon las características de Python en cuanto al manejo de programación sobre objetos. Esta forma de organizar el código es muy "elegante", permite tener una mejor clasificación de la información que manejamos, además de que se pueden incluir funciones que realicen operaciones internas por cada objeto. Sin embargo, cada metodología tiene sus "pros y contras", por lo que usar objetos puede ser visualmente mejor en el código y en la estructuración de éste, pero el costo que representa hacer esto se refleja en la gestión de memoria y el tiempo utilizado. Al definir objetos que contienen diversas variables de distintos tipos, funciones, vectores, matrices, etcétera; la computadora acomoda la memoria de tal manera que cada objeto está almacenado linealmente y al momento de que se utilizan variables de distintos objetos, la computadora necesita revisar grandes espacios de memoria para localizar quizá solo un dato. Esto es ineficiente en el punto de vista de las memorias de las computadoras, ya que se podría almacenar un solo vector con esas variables a utilizar y la computadora solo gestionaría la información con una memoria contigua y más rápido. Pero al trabajar con objetos propios del modelador **Dynamo**, se decidió utilizar la programación de objetos en algunos puntos clave para dar mejor entendimiento del código y clasificar mejor la información, tomando en cuenta que habrá un decremento en la eficiencia de memoria.

Lo primero que hace este algoritmo es identificar las barras perimetrales de cada losa, este procedimiento quizá es de los más lentos, debido a que en cada nivel buscará cuales barras son la que tienen intersección con la superficie. Posteriormente solo se verifica cuantas vigas definen la losa. Si son cuatro

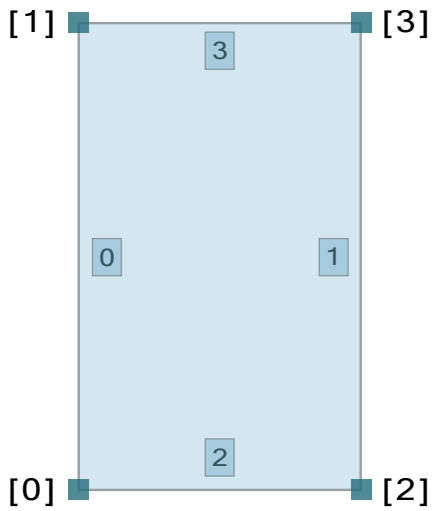


Figura 3.70: Referencia de nodos en losas. Se muestra el alineamiento de nodos y barras utilizado en el código para generar las losas tributarias.

barras, entonces se empleará la metodología de distribución para losas rectangulares mediante triángulos y trapecios; mientras que si el número de barras perimetrales es distinto, se utilizará la metodología del polígono que localiza el centroide de la figura para obtener las áreas tributarias.

Si se activa el caso de losas rectangulares, lo primero es utilizar los puntos extremos de la losa y con la ayuda de la función `slab_align(vert_nodes)` acomoda los puntos de acuerdo las relaciones matemáticas que se emplearán para la distribución de losas rectangulares, los nodos deben estar acomodados de acuerdo a la imagen 3.70. Esta función lo primero que hace es determinar los nodos máximo y mínimo en el espacio geométrico. Para la obtención del nodo [0], se tiene que encontrar el nodo más negativo de la losa con comparaciones en X y Y. Siguiendo la misma lógica para el nodo [1] se realiza la comparación únicamente en X debido a que el nodo [0] ya fue localizado; en el nodo [2] la comparación es con Y, y al final el último nodo pasa a ser el [3]. Durante cada localización hay que borrar cada nodo encontrado y volver a localizar máximos y mínimos. De esta manera ya tenemos los nodos listos para generar las líneas de las áreas tributarias.

El siguiente paso dentro de la metodología de losas rectangulares es la utilización de la función `get_tribu_areas_rec_surf(vert_nodes_aligned,i)` la cual se encarga de generar las áreas tributarias. La función requiere de los nodos previamente alineados según la referencia para crear unos nodos que definen las direcciones de las líneas perimetrales. Se calculan las longitudes de las líneas y mediante una comparación de distancias, se obtiene la dirección en la que los trapecios están dirigidos, en torno al eje X o al eje Y. Por ello se necesita conocer el lado largo de la losa y el lado corto. Aunque el procedi-

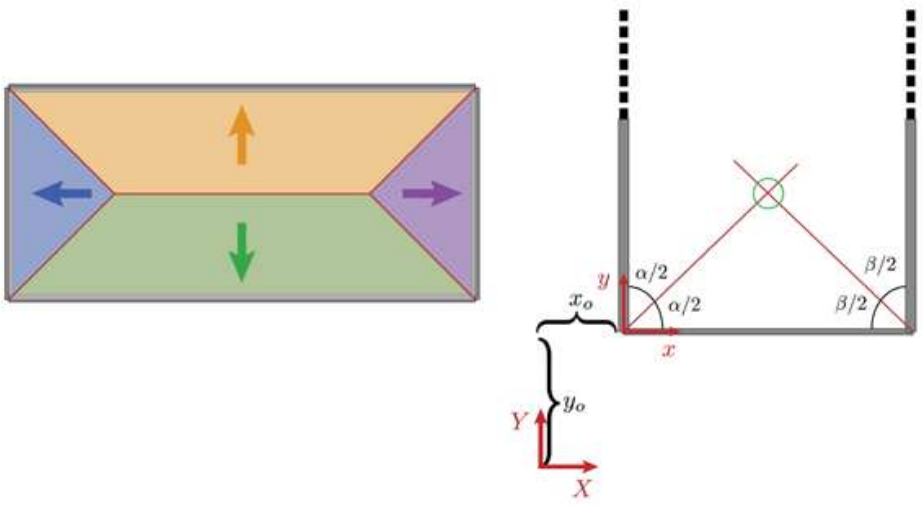


Figura 3.71: Distribución de cargas en losas rectangulares. Esta figura muestra un caso de las posibles relaciones matemáticas necesarias para generar las áreas tributarias triangulares y trapezoidales.

miento es el mismo para los dos casos, es importante hacer la separación para utilizar los vectores indicados. Entonces, de acuerdo a la figura mostrada 3.70, se calculan los ángulos entre las caras de la superficie y como podemos ver se necesita de dos puntos principalmente para generar las líneas que componen la áreas. Como se comentó antes, el método geométrico utiliza líneas que parten de las esquinas y con una inclinación de 45° forman una intersección la cual denota un vértice de las áreas tributarias. Debido a que se desea automatizar este procedimiento, se necesito de relaciones matemáticas trigonométricas para calcular las coordenadas de los dos puntos vértice necesarios. La forma en que trabaja el algoritmo es calculando el ángulo necesario por cada vértice, y graficando la solución del sistema de ecuaciones 3.2 y 3.3. Dicha solución describe los dos puntos internos, nombrados como "inferior" y "superior". Con estos puntos internos y los puntos extremos alineados de las superficies se crean conjuntos de líneas para generar las áreas tributarias. Tres líneas para el conjunto de triángulos y cuatro para el conjunto de trapecios. En el caso donde la dirección del lado corto y lado largo varían se tuvo que tener cuidado con el conjunto de líneas y los ángulos calculados, ya que el sistema es similar solamente se debe cambiar la dirección y sistema coordenado. También existe un tercer caso para estas losas, y es cuando los lados cortos y largos son iguales, es decir, la losa es cuadrada. Para ello solo se calculo un punto central y el conjunto de líneas siempre serán triángulos como se observa en la imagen 3.64 de la sección anterior.

$$y = tan(\alpha/2)x \tag{3.2}$$

$$y = -tan(\beta/2)x + b \tag{3.3}$$

El sistema de ecuaciones 3.2 y 3.3 denota la formulación de la recta utilizando el ángulo respecto al eje X partiendo del nodo [0] y el nodo [2] respectivamente. La solución corresponde al caso más general para la obtención del punto inferior de una losa rectangular, ya que para el punto superior se debe cambiar la dirección de la recta. Sin embargo las condiciones para la solución variarán de acuerdo a la posición de la losa. Un ejemplo es cuando el lado largo de la losa tiene la dirección del eje X, entonces las ecuaciones se modifican. Además cuando las losas tienen cuatro lados pero no son perfectos rectángulos también se utiliza esta metodología, sin embargo hay modificaciones en los signos de las ecuaciones. La figura 3.72 muestra las posibles combinaciones de losas de cuatro lados a considerar.

Si el algoritmo identifica que la superficie de la losa no es rectangular, es decir que el número de lados difiere de cuatro, entonces se activa la función de losas poligonales, se muestran algunas opciones en la figura 3.73. Este procedimiento comienza utilizando las barras perimetrales para obtener los puntos extremos, los cuales se verifican para evitar que existan duplicados. Se almacenan en los objetos que antes se mencionaron y posteriormente se crean los conjuntos de líneas que describen las áreas tributarias triangulares. Este proceso es por mucho más sencillo que las losas de cuatro lados, debido a que el único punto auxiliar es el centroide pero como recordaremos, el centroide de cada superficie ya lo tenemos disponible en un arreglo. Para conocer el procedimiento de la obtención del centroide, véase la sección 3.3.2 de este mismo capítulo. Durante este procedimiento se emplean funciones dentro de cada objeto losa, que calculan el área mediante la función de **Dynamo**, `Surface.Area()` y este valor se convierte a cm^2 que es la unidad a utilizar en el software **MECA**. Al finalizar esta función solamente se guardan algunos vectores con información necesaria a retornar, que será útil para el análisis estructural, como por ejemplo las líneas perimetrales y las magnitudes de áreas en unidades correspondientes. Además se implementó un visualizador de distribución de áreas tributarias, con ayuda de un rango de colores. Entonces de acuerdo a la magnitud de carga se hizo una interpolación de color, la carga mínima en el modelo tendrá un color azul mientras la carga máxima tendrá un color rojo. Por lo tanto también se retornan las superficies de las áreas tributarias y el valor de color que se utilizará más adelante para la visualización.

La siguiente función que se ejecuta en el espacio principal, se denomina `search_count _beams(beams, beams_index)`, la cual se encarga de buscar en cada superficie de losa las vigas que la rodean. Este procedimiento es de los más lentos durante la ejecución del módulo ***Structure*** debido a que se debe buscar entre todas las barras de cada nivel la posibilidad de pertenecer a una losa en específico. Para implementar esto, se decidió utilizar las funciones de permutación que ofrece Python en la librería `itertools`. La manera de hacer esto, es utilizando los puntos extremos de cada losa y obteniendo las permutaciones de

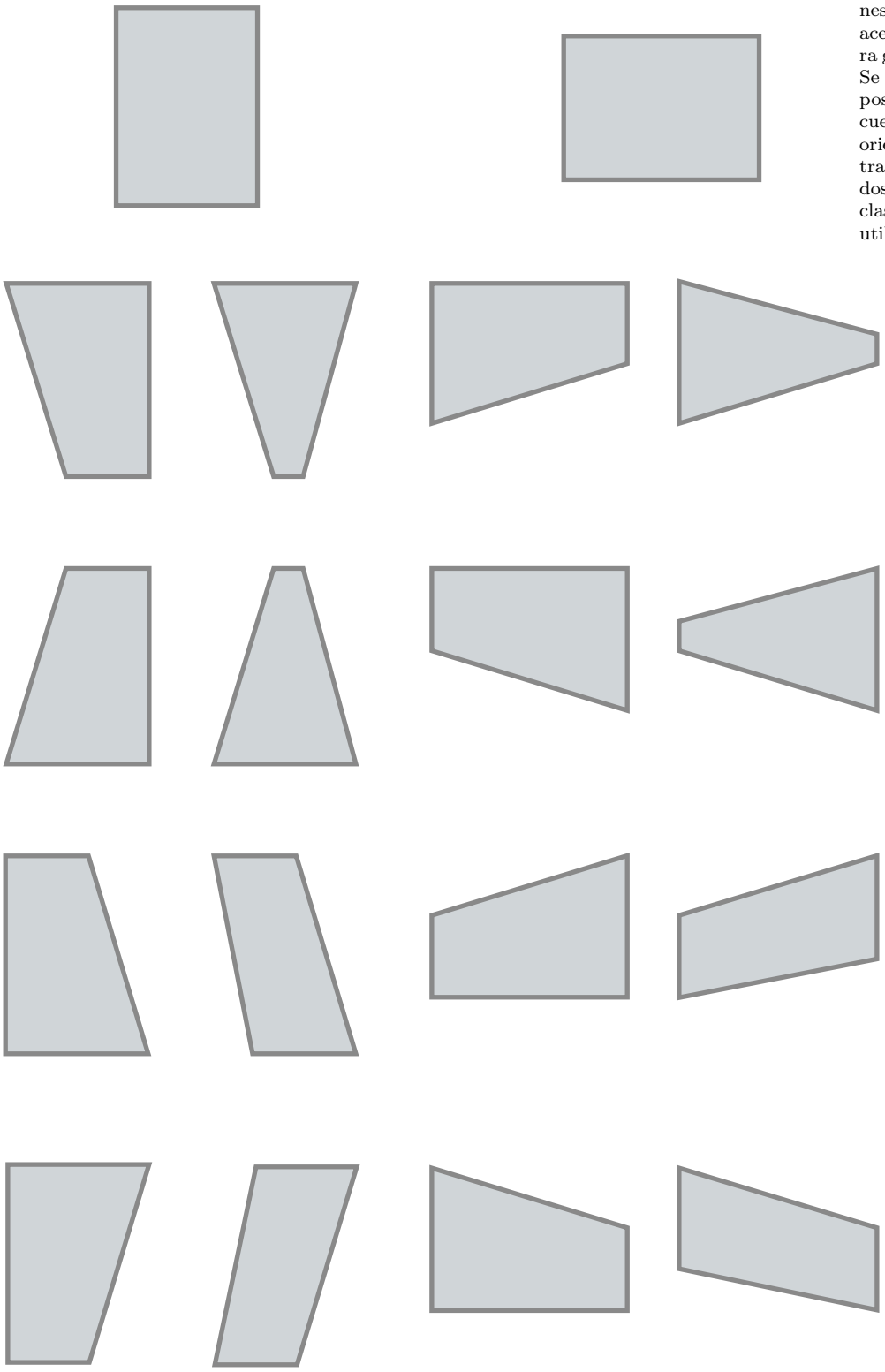


Figura 3.72: Losas rectangulares. Posibles combinaciones de losas de cuatro lados aceptadas por el módulo para generar las áreas tributarias. Se observan dos grandes grupos, uno donde las vigas se encuentran con los lados largos orientados verticalmente mientras el otro grupo tienen sus lados largos horizontales. Esto se clasifica así debido al método utilizado.

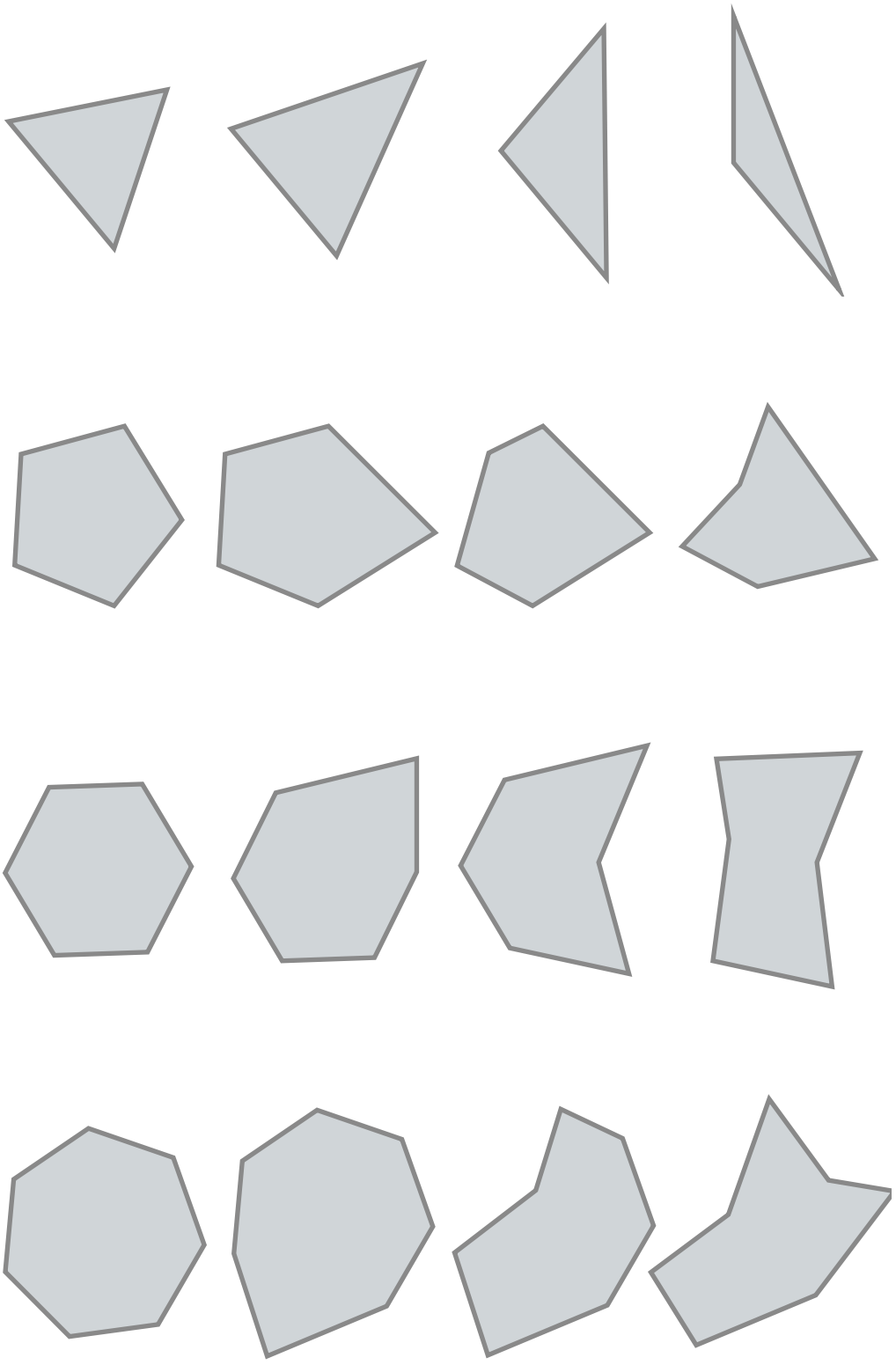


Figura 3.73: Losas poligonales. Algunos ejemplos de losas que cumplen con la clasificación de losas poligonales, en las cuales se utiliza su centroide para generar las áreas tributarias. Se muestran polígonos regulares como base (triángulo, pentágono, hexágono, heptágono) y se modifica su geometría para generar polígonos irregulares en base a estos.

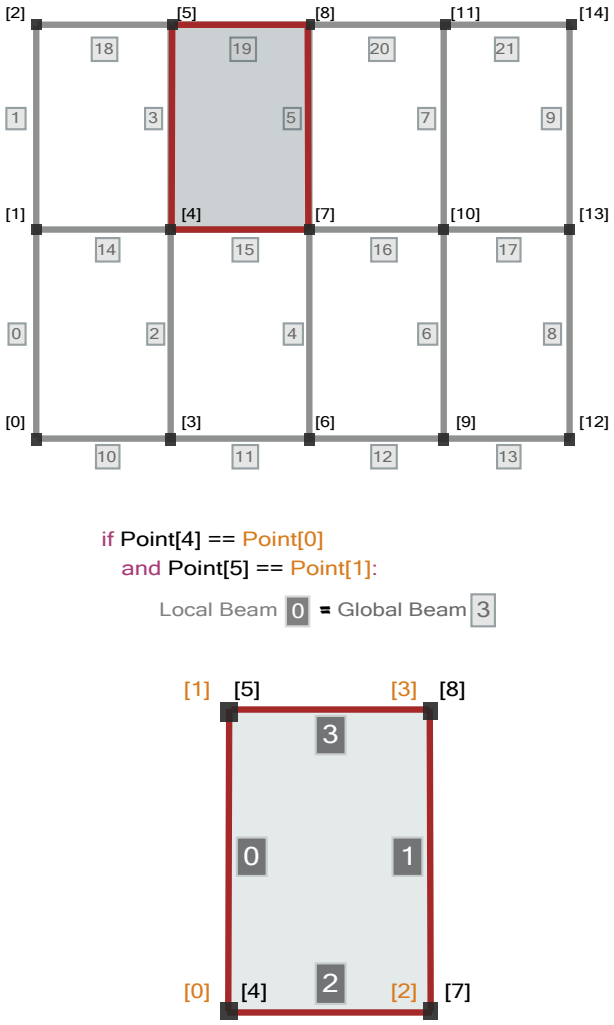


Figura 3.74: Búsqueda de vigas en losas. Se muestra una edificación rectangular con la numeración de vigas y puntos global, y además se muestra una losa con la numeración local. Se describe a grandes rasgos el condicional utilizado para relacionar las vigas con índices globales y las vigas perimetrales con índices locales.

sus nodos en arreglos, posteriormente se verifica mediante condicionales si las coordenadas del vector de permutación corresponde a alguna barra, si es punto inicial o final. Si da como positiva esta comparación, entonces se guarda un vector con los índices de cada barra que pertenece a cada losa y de esa manera ya sabemos como están conectadas las barras con las superficies. Una figura que a grandes rasgos muestra este planteamiento es la imagen 3.74, incluso se ilustra una comparación de nodos para conocer la relación local-global de las vigas.

Ya conociendo las barras que soportan cada losa, el último paso es asignar las cargas. Por lo que se implementó la función `load_assign(beams,index_frame_vec, slab_floor_vec)` la cual retorna un vector con la misma posición que las barras, pero con la magnitud de carga que se considerará para el análisis

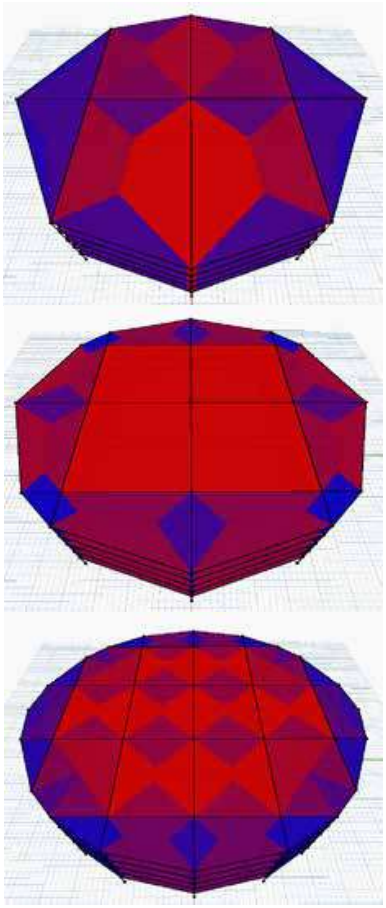


Figura 3.75: Resultado de distribución de fuerzas. Modelo circular sometido al módulo de estructuración y distribución de cargas. El algoritmo de distribución identifica el tipo de losa por lo que la primer imagen muestra triángulos y losas de cuatro lados, la segunda solamente losas de cuatro lados y la tercera tiene más diversidad de figuras.

estructural. Para ello solamente se requiere de ciclos anidados que recorren los índices de barra, las superficies y las barras que las sostienen. Se hace una verificación del índice de barras con el índice de las barras que soportan cada segmento de los o área tributaria. Y de esta manera se asigna una carga uniformemente repartida utilizando la magnitud del área tributaria calculada antes, multiplicando por la magnitud de carga superficial proporcionada por el usuario y dividiendo la longitud de cada barra. Esta metodología de distribución de cargas se explica en la sección 3.6.1 y se emplea la utilización del método simplificado por las razones mencionadas antes. Los valores que regresa el módulo ***Slab Loads*** son primeramente los parámetros de módulos anteriores para dar continuidad con la conexión de información que se ha utilizado desde el principio, posteriormente se regresa un arreglo con la información para la visualización por colores, un arreglo con las cargas uniformemente distribuidas por barra y por último los índices de vigas por losa.

Figura 3.76: El Tajín. "La Pirámide de los Nichos", obra de la cultura Totonaca (Veracruz, México), es considerada como uno de los centros religiosos más importantes de Mesoamérica (12).

3.7. Análisis Estructural

Hasta ahora siguiendo con la descripción de los módulos ordenados según se van conectando en el diagrama *DAG*, ya generamos un sólido tridimensional, ya se obtuvieron los niveles deseados en las funciones de ***Floors*** (sección A.2) y además el usuario ya determinó la configuración estructural mediante el módulo ***Grid*** (sección 3.3); ya se generó la estructuración de la edificación de acuerdo a los parámetros empleados en el módulo ***Structure*** (sección 3.5) utilizando los catálogos del software **MECA** en las funciones de lectura de materiales (sección 3.4), y además se asignaron fuerzas superficiales distribuidas en losas en el módulo ***Slab Loads*** (sección 3.6). Por lo que único que resta es realizar el análisis estructural y conocer el comportamiento de la edificación bajo cargas gravitacionales.

3.7.1. Breve Historia del Análisis de Estructuras

Desde que el humano necesito construir para disponer de un lugar seguro debido a los factores climatológicos adversos o para cuidar de su integridad debido a los depredadores, los sistemas constructivos han evolucionado exponencialmente. Los tipos de materiales usados, la disposición de los elementos portantes y las fases constructivas han sido mejorados mientras se conoce con más detalle el comportamiento de éstos. Grandes culturas antiguas lograron dejar su huella en el mundo con increíbles construcciones de múltiples usos, y aunque se basaron en un inicio en el empirisimo, poco a poco utilizaron una lógica basada en matemáticas y física para desarrollar construcciones más complejas.

No fue hasta que Galileo Galilei comienza con un estudio más sistemático del análisis estructural, mediante su obra *Los discursos sobre dos nuevas ciencias* en 1638, ya que en estos escritos se da inicio a la etapa de desarrollo de la *Mecánica de Materiales* la cual es fundamental para el *Análisis Estructural.* Posterior a ese momento, se hicieron desarrollos notables en este campo prometedor en donde destacan algunos científicos como Robert Hooke, Sir Isaac Newton, John Bernoulli, Leonhard Euler y Carles-Augustin Coulomb. Basados en sus contribuciones, en el siglo XIX se desarrollan nuevas técnicas (ahora llamados *métodos clásicos*) con los que se inicia la mecánica de materiales, entre éstos *el método de la viga conjugada, círculos de Mohr para esfuerzo y deformación, el teorema del trabajo mínimo de Castigliano, el método de distribución de momentos de Cross y el método pendiente-deflexión de Maney.* Pero hasta la década de los 1950's comenzó a aparecer la literatura especializada en el *Análisis Matricial de Estructuras*, la cual tuvo un inicio lento debido a que los ingenieros de esa época no estaban tan familiarizados con el *Algebra Matricial*, sino que tenían una mayor experiencia en métodos de operaciones iterativas como lo es el Método de Cross. Sin embargo existieron dos circunstancias que propociaron el florecimiento de los métodos matriciales. La primera fue que las estructuras de esa época comenzaban a ser más complejas, y por consecuencia el análisis requerido se incrementó al grado de no poder obtener información suficiente para un diseño óptimo, por ello era necesario implementar métodos más precisos. Por otro lado, la *Era de la Computación* ya había comenzado y con éstos equipos la solución de grandes sistemas de ecuaciones simultáneas, que implicaban semanas de trabajo, ahora se podían realizar en cuestión de minutos. En conjunto a la necesidad de estos análisis y la capacidad de realizar cálculos numéricos de alta velocidad, se realizó un extenso desarrollo en la investigación de métodos matriciales. Aunque inicialmente se desarrollo la formulación matricial del *Método de la Fuerza Redundante*, poco después apareció el *Método de las Rigideces* (también conocido como el método de los desplazamientos). Aunque ahora ya existen otros métodos matriciales más precisos y potentes para problemas de análisis muy específicos, como lo es el "Método de los Elementos Finitos" y sus variantes modernas, el *Método Matricial de las Rigideces* sigue estando presente en todos los programas comerciales de análisis estructural.(42).

3.7.2. Método Matricial de las Rigideces

Para el análisis de estructuras se requiere de formulaciones matemáticas que aproximen los modelos a la realidad, las cuales necesitan de una idealización para su cálculo. El objetivo es conocer los elementos mecánicos actuantes que servirán posteriormente para su verificación mediante códigos de construcción por zona geográfica, que garantizan la seguridad del diseño propuesto. La simplificación geométrica utilizada en el *método matricial de las rigideces* es la

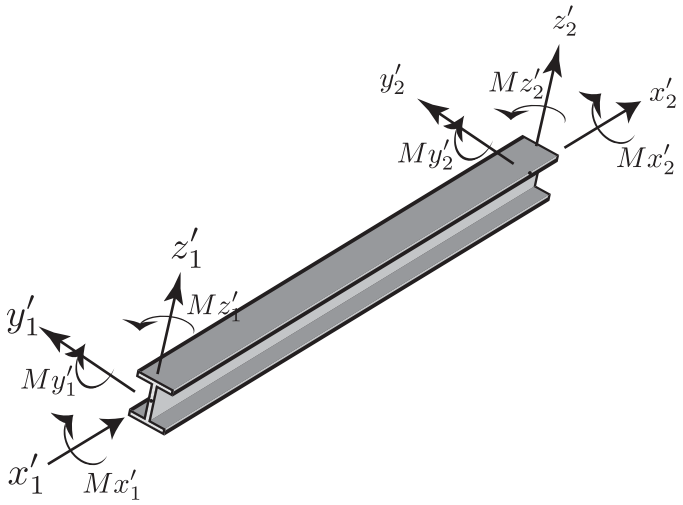


Figura 3.77: Fuerzas en Extremo Local. Barra de tipo *marco tridimensional*, la cual tiene en sus extremos las fuerzas impuestas en un sistema coordenado local (42).

consideración de barras prismáticas como líneas idealizadas y colocadas en el centroide de cada barra, las uniones se denominan *nodos* y son representados como puntos que conectan las barras del modelo. Los nodos están relacionados directamente con los *grados de libertad*, siendo éstos los posibles desplazamientos tanto lineales como rotacionales. La *rigidez* dentro del análisis estructural se entiende como el valor numérico que relaciona las deformaciones que sufre un cuerpo con los esfuerzos que soporta, y está relacionado con el comportamiento lineal que presentan los materiales utilizados para construir los elementos estructurales (columnas y trabes). Entonces para relacionar los grados de libertad que posee una barra y su rigidez, es necesario construir una *matriz elemental*, que como su nombre lo indica, es una matriz por cada barra en el modelo estructural. Para conocer el comportamiento de la estructura en conjunto se construye una *matriz de rigidez ensamblada*, la cual relaciona a todas las barras con los grados de libertad y las fuerzas implicadas en el análisis. La solución del sistema de ecuaciones dado por la matriz ensamblada es un vector con los resultados numéricos que representan los desplazamientos que sufre la estructura. Estos desplazamientos son sustituidos en las matrices elementales, y con esto se obtienen las fuerzas en cada barra. Una de las consideraciones en el método es el empleo de diferentes sistemas coordenados, uno local por cada barra y uno global de la estructura, es por ello que deben de considerarse transformaciones matriciales para vincular estas dos referencias (42).

Existen diversas matrices elementales según los grados de libertad deseados en nuestro elemento estructural. Para los fines de la programación de un módulo que permita realizar un análisis estructural en un modelo paramétrico 3D será necesario utilizar la formulación más completa para una barra mediante

el método matricial, ésta es la matriz elemental para estructuras denominadas *marcos tridimensionales*. Su expresión en el sistema coordenado local se muestra en la ecuación 3.4 (38), la figura 3.77 muestra este elemento estructural con las fuerzas en extremo aplicadas en los nodos en el sistema local.

$$
\begin{bmatrix} f'_{1x} \\ f'_{1y} \\ f'_{1z} \\ m'_{1x} \\ m'_{1y} \\ m'_{1z} \\ \hline f'_{2x} \\ f'_{2y} \\ f'_{2z} \\ m'_{2x} \\ m'_{2y} \\ m'_{2z} \end{bmatrix} =
\left[\begin{array}{cccccc}
\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{12EI_{zz}}{L^3} & 0 & 0 & 0 & \frac{6EI_{zz}}{L^2} \\
0 & 0 & \frac{12EI_{yy}}{L^3} & 0 & -\frac{6EI_{yy}}{L^2} & 0 \\
0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\
0 & 0 & -\frac{6EI_{yy}}{L^2} & 0 & \frac{4EI_{yy}}{L} & 0 \\
0 & \frac{6EI_{yy}}{L^2} & 0 & 0 & 0 & \frac{4EI_{zz}}{L} \\ \hline
-\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{12EI_{zz}}{L^3} & 0 & 0 & 0 & -\frac{6EI_{zz}}{L^2} \\
0 & 0 & -\frac{12EI_{yy}}{L^3} & 0 & \frac{6EI_{yy}}{L^2} & 0 \\
0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\
0 & 0 & -\frac{6EI_{yy}}{L^2} & 0 & \frac{2EI_{yy}}{L} & 0 \\
0 & \frac{6EI_{yy}}{L^2} & 0 & 0 & 0 & \frac{2EI_{zz}}{L}
\end{array}\right] \cdots
$$

$$
\cdots
\left[\begin{array}{cccccc}
-\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{12EI_{zz}}{L^3} & 0 & 0 & 0 & \frac{6EI_{zz}}{L^2} \\
0 & 0 & -\frac{12EI_{yy}}{L^3} & 0 & -\frac{6EI_{yy}}{L^2} & 0 \\
0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\
0 & 0 & \frac{6EI_{yy}}{L^2} & 0 & \frac{2EI_{yy}}{L} & 0 \\
0 & -\frac{6EI_{zz}}{L^2} & 0 & 0 & 0 & \frac{2EI_{zz}}{L} \\ \hline
\frac{EA}{L} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{12EI_{zz}}{L^3} & 0 & 0 & 0 & \frac{6EI_{zz}}{L^2} \\
0 & 0 & \frac{12EI_{yy}}{L^3} & 0 & \frac{6EI_{yy}}{L^2} & 0 \\
0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\
0 & 0 & \frac{6EI_{yy}}{L^2} & 0 & \frac{4EI_{yy}}{L} & 0 \\
0 & -\frac{6EI_{zz}}{L^2} & 0 & 0 & 0 & \frac{4EI_{zz}}{L}
\end{array}\right]
\begin{bmatrix} u'_1 \\ v'_1 \\ w'_1 \\ \mu'_1 \\ \psi'_1 \\ \omega'_1 \\ \hline u'_2 \\ v'_2 \\ w'_2 \\ \mu'_2 \\ \psi'_2 \\ \omega'_2 \end{bmatrix} \tag{3.4}
$$

En donde el sistema de ecuaciones está en función de propiedades geométricas de la barra (A, L, I_{zz}, I_{yy}, J) y propiedades del material utilizado (E, G). Se puede observar la relación que hay entre los desplazamientos lineales (u', v', w_n) y rotacionales (μ', ψ', ω_n) que sufre el elemento estructural, respecto a las fuerzas y momentos aplicados ($f'_{n\ x,y,z}$, $m'_{n\ x,y,z}$). Debido a que la mayoría de las estructuras cuentan con más elementos estructurales que solo una viga, es necesario trasladar la formulación matricial anterior a un sistema global en donde puedan interactuar incógnitas de todas las barras que estén incluidas

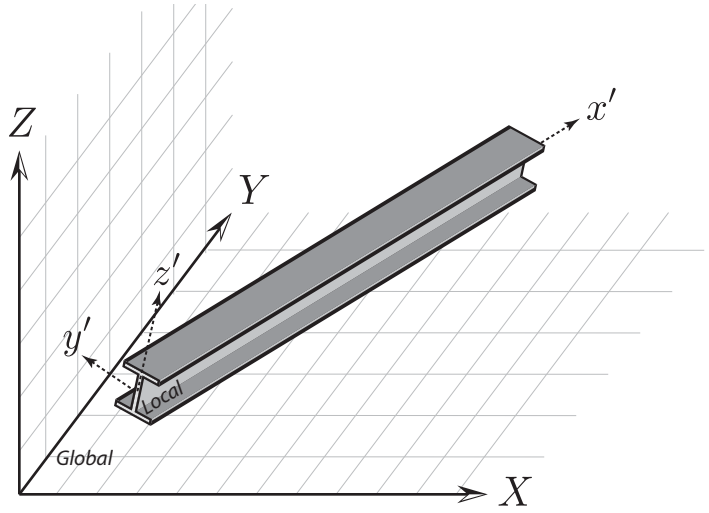


Figura 3.78: Sistemas Local-Global. Viga que muestra la relación entre el sistema local de una barra con respecto a un espacio denotado como sistema global (42).

en el análisis. Para esto es necesario hacer un desplazamiento de coordenadas y una rotación a cada barra de la estructura, dicha transformación matricial para el cambio de coordenadas puede hacerse de varias maneras pero se utilizará un procedimiento basado en una rotación mediante un vector auxiliar que describe el plano que coincide con el eje de la sección transversal. Utilizando un vector dirigido a lo largo del elemento transversal $\widehat{v}_x$ y otro vector del origen al punto auxiliar denotado como v_a en la figura 3.79, es posible conocer un vector perpendicular al plano formado mediante un producto cruz (ec. 3.5):

$$\overline{v}_z = \overline{v}_a \times \widehat{v}_x = \begin{vmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ v_{ax} & v_{ay} & v_{az} \\ v_{xx} & v_{xy} & v_{xz} \end{vmatrix} \tag{3.5}$$

Normalizando el vector $\overline{v}_z$ obtenemos:

$$\widehat{v}_z = \frac{v_x}{\sqrt{v_{zx}^2 + v_{zy}^2 + v_{zz}^2}}\hat{\imath} + \frac{v_y}{\sqrt{v_{zx}^2 + v_{zy}^2 + v_{zz}^2}}\hat{\jmath} + \frac{v_z}{\sqrt{v_{zx}^2 + v_{zy}^2 + v_{zz}^2}}\hat{k} \tag{3.6}$$

Para encontrar el tercer y último eje local, emplearemos el mismo procedimiento que en el paso anterior, calcularemos el producto cruz entre dos vectores, como ya contamos con el eje local $\widehat{v}_x$ y el eje local $\widehat{v}_z$, entonces el vector perpendicular a estos dos será $\widehat{v}_y$:

$$\widehat{v}_y = \widehat{v}_z \times \widehat{v}_z = \begin{vmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ v_{xx} & v_{xy} & v_{xz} \\ v_{zx} & v_{zy} & v_{zz} \end{vmatrix} \tag{3.7}$$

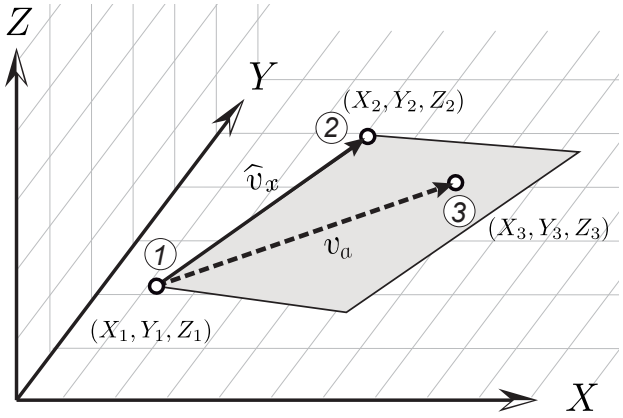


Figura 3.79: Matriz de Transformación. Procedimiento utilizado para realizar la transformación de coordenadas del sistema local al global (42).

Ya contamos con los ejes locales de la barra, por lo tanto podemos organizar sus componentes en forma matricial, de manera que se relacionen los vectores con dirección al eje global $\boldsymbol{X}$y con dirección al eje local $\boldsymbol{X'}$:

$$\begin{bmatrix} \widehat{v}_x \\ \widehat{v}_y \\ \widehat{v}_z \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ z'_1 \end{bmatrix} = \begin{bmatrix} v_{xx} & v_{xy} & v_{xz} \\ v_{yx} & v_{yy} & v_{yz} \\ v_{zx} & v_{zy} & v_{zz} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \tag{3.8}$$

Así llegamos a una expresión generalizada de la matriz de transformación:

$$\boldsymbol{f'_x} = \boldsymbol{r}\,\boldsymbol{f_x} \tag{3.9}$$

Sin embargo, como podemos ver en la ecuación 3.8, esta matriz es de tamaño [3x3]; y ya que la matriz de rigidez local del elemento marco tridimensional (ec. 3.4) tiene un tamaño de [12x12], llenaremos una matriz que denominaremos $\boldsymbol{R}$ con cuatro submatrices $\boldsymbol{r}$:

$$\boldsymbol{R} = \left[\begin{array}{ccc|ccc|ccc|ccc} v_{xx} & v_{xy} & v_{xz} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{yx} & v_{yy} & v_{yz} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v_{zx} & v_{zy} & v_{zz} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & v_{xx} & v_{xy} & v_{xz} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & v_{yx} & v_{yy} & v_{yz} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & v_{zx} & v_{zy} & v_{zz} & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & v_{xx} & v_{xy} & v_{xz} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v_{yx} & v_{yy} & v_{yz} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v_{zx} & v_{zy} & v_{zz} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & v_{xx} & v_{xy} & v_{xz} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & v_{yx} & v_{yy} & v_{yz} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & v_{zx} & v_{zy} & v_{zz} \end{array}\right] \tag{3.10}$$

Utilizando esta matriz se hacen unas sustituciones que se omitirán en este apartado, pero al final se obtiene una matriz tipo *marco tridimensional* en el sistema global denotada por la transformación de la matriz local 3.4:

$$\begin{bmatrix} \boldsymbol{F_{1x}} \\ \boldsymbol{M_{1x}} \\ \boldsymbol{F_{2x}} \\ \boldsymbol{M_{2x}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r^T}\, k'_{11}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{21}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{31}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{41}\, \boldsymbol{r} \\ \boldsymbol{r^T}\, k'_{12}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{22}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{32}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{42}\, \boldsymbol{r} \\ \boldsymbol{r^T}\, k'_{13}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{23}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{33}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{43}\, \boldsymbol{r} \\ \boldsymbol{r^T}\, k'_{14}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{24}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{34}\, \boldsymbol{r} & \boldsymbol{r^T}\, k'_{44}\, \boldsymbol{r} \end{bmatrix} \begin{bmatrix} \boldsymbol{U_1} \\ \boldsymbol{\mu_1} \\ \boldsymbol{U_2} \\ \boldsymbol{\mu_2} \end{bmatrix} \tag{3.11}$$

Donde cada elemento $\boldsymbol{r}$ es la matriz de transformación, $\boldsymbol{r^T}$ corresponde a la misma matriz de transformación pero transpuesta y los elementos k'_{ij} corresponden a los elementos de la matriz local del elemento marco tridimensional, representados por sub-matrices. Posteriormente, al tener calculadas las matrices de rigidez correspondientes a cada elemento en el sistema de coordenadas global 3.11 de la estructura, se prosigue a realizar el ensamblaje de acuerdo a los grados de libertad que no están restringidos por condiciones de apoyo. El sistema a resolver puede ser tan grande que la necesidad de utilizar un equipo de cómputo potente se vuelve importante, es justo este paso donde la importancia de las computadoras en el análisis estructural se vuelve grande. La forma de solucionar este sistema de ecuaciones se hace mediante *métodos numéricos* especializados en sistemas matriciales de gran tamaño, se debe considerar aspectos como lo es la rapidez de cálculo debido a la memoria utilizada, y esto se plantea mediante la implementación de un código eficiente que utilice recursos computacionales óptimos. De forma general las ecuaciones a resolver se representan con el siguiente arreglo matricial:

$$\big[\mathbf{K}\big]\ \big[\mathbf{D}\big] = \big[\mathbf{F}\big] \tag{3.12}$$

Donde:

- **K**: Matriz de rigidez ensamblada.
- **F**: Vector de fuerzas en sistema de coordenadas global.
- **D**: Vector de desplazamientos. Solución del sistema lineal de ecuaciones.

La solución numérica del sistema mostrado en la ecuación 3.12, da como resultado un vector de desplazamientos **D** por cada grado de libertad que existe en sistema estructural, entonces el paso restante es sustituir esos valores en la matriz elemental global (ec. 3.11) de cada barra en el correspondiente grado de libertad y de esta manera se obtienen las fuerzas en extremo en el sistema coordenado global como se muestra en la imagen 3.80.

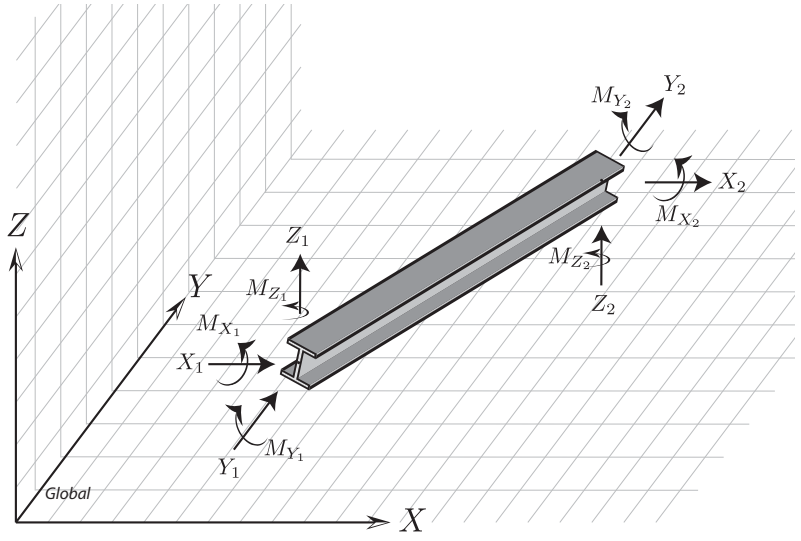


Figura 3.80: Fuerzas en Extremo Global. Barra de tipo *marco tridimensional*, la cual tiene en sus extremos las fuerzas resultantes del análisis en un sistema coordenado global (42).

Los desplazamientos producto de la solución numérica del sistema de ecuaciones ensamblado son empleados para conocer el comportamiento estructural en conjunto como por ejemplo para graficar una estructura deformada. Mientras que las fuerzas resultantes del análisis son las utilizadas para el siguiente paso en el proceso de Ingeniería Estructural, que es realizar cálculos específicos por cada tipo de material en los que se verifica la resistencia de éste debido a las acciones impuestas. Este proceso es comúnmente llamado *Diseño Estructural*, y lo que regularmente devuelve es un valor de eficiencia de la barra en porcentaje donde 0 % es una barra sin cargas y sobre-dimensionada, una barra entre 70 % y 85 % esta en un rango óptimo y útil para soportar las fuerzas, mientras que eficiencias mayores no son recomendadas debido a que el elemento estructural comenzará a tener fallas.

3.7.3. Extracción de Información para el Análisis Estructural

Como se vio en la sección 3.7.2, el análisis matricial de estructuras depende de varios parámetros que son implicados dentro de los sistemas de ecuaciones que necesitan resolverse para conocer los desplazamientos que sufre una estructura debido a una carga impuesta. Entre lo que podemos resaltar, es necesario proponer una configuración donde la topología de la estructura esté bien definida, esto se refiere a definir una numeración especial para cada barra, además de una numeración para cada nodo y con esto se debe de conocer cómo y con qué entidades se conecta cada elemento estructural. Se deben definir propiedades geométricas, las cuales se proponen en cuanto a la sección transversal que se utilizará para el diseño; las propiedades mecánicas de los materiales empleados también deben estar presentes como pueden ser los módulos de elasticidad normal y cortante. También se deben asignar condiciones y/o restricciones sobre nodos que representen los apoyos de la estructura. Con estos datos de entrada

es posible calcular, ensamblar y resolver las ecuaciones representadas en forma matricial.

Para realizar el análisis estructural se utilizará un software desarrollado por investigadores de la Universidad de Guanajuato y el Centro de Investigación en Matemáticas, el cual se hablará en la siguiente sección que tiene por número 3.7.4. Por ahora solo se mencionará que la forma de ejecutar el análisis es mediante un archivo de entrada (.dat) que se generará en el módulo programado en Python, denominado ***Data_Input_Meca***, el cual se muestra en el apéndice A.8. Para ello se deberá gestionar y organizar la información que se obtuvo en los módulos anteriores. La imagen muestra un ejemplo de los datos que son necesarios para ejecutar el análisis.

Como primer paso de este módulo, se comienza con algunos letreros que el software necesita leer, como lo son el número de versión del programa, el nombre del proyecto, las unidades de longitud y las unidades de fuerza. Posteriormente se necesita el número de nodos, entonces se necesita de la información generada por el módulo ***Structure*** para conocer este valor. De igual manera se hace un conteo de barras, primero con las vigas y luego con las columnas que se encuentran en la estructura. Después se pide el número de restricciones de desplazamiento en la estructura, **MECA** lo denomina número de "prescripciones", para ello se utilizará únicamente el número de nodos que se encuentran en la base de la edificación. El número de casos de carga se dejó fijo y con el valor de 1, el tipo de estructura se utilizó como empotrada - empotrada, el número de grados de libertad por nodo es de 6 debido a que se está trabajando en un espacio tridimensional. En cuanto al número de materiales, se utilizó el tamaño del arreglo proporcionado por el módulo de materiales de la sección 3.4. Se pedirá que la escritura de información se escriba en res pero se omitirá la escritura de la matriz de rigidez. El número de dimensiones es 3, no se considerarán apoyos elásticos ni inclinados. De inicio tampoco se utilizará el módulo de optimización que ofrece **MECA**, y el problema es estático.

El siguiente punto de la hoja de entrada mostrada en la figura 3.81, es la *Definición de Barras*. Aquí se define la topología de la estructura, ya que se indican los índices de los nodos que componen cada barra. Esta información ya la obtuvimos en las funciones descritas en el módulo ***Structure*** de la sección 3.5, por lo que simplemente se debe vaciar los datos mediante ciclos en la hoja. Debido a que antes se hizo la clasificación de barras entre vigas y columnas, primero se necesitará escribir la información de una clasificación y posteriormente de la otra. El primer elemento es el número de material empleado en la barra, ese dato lo obtenemos del módulo de materiales. El segundo elemento es el nodo inicial, posteriormente el nodo final. El valor que sigue da a conocer al programa la forma en que se calculará la matriz de rotación para el elemento marco, si es un cero se utilizará un solo ángulo de rotación sobre la sección

Figura 3.81: Información Estructural. Ejemplo de datos necesarios para realizar un análisis estructural mediante el software **MECA**, se muestra un archivo tipo .DAT .

```
PreMECA_____________________________= 1.0
Proyecto____________________________= Dynamo
 Unidadeslongitud___________________= Cm,Kg,Seg
 Unidades_fuerza____________________= Cm,Kg,Seg
 Numero_de_nodos____________________= #
 Numero_de_elementos________________= #
 Numero_de_prescripciones___________= #
 Numero_de_casos_de_carga___________= #
 Tipo_de_estructura__art,_emp_o_comb____= #
 Numero_de_grados_de_libertad_por_nodo__= #
 Numero_de_materiales_______________= #
 Escritura_dat_en_res_____________0_o_1__= #
 Escritura_de_mat_de_rigid_______0_o_1__= #
 Numero_de_dimensiones____________2_o_3__= #
 Numero_de_apoyos_elasticos_________= #
 Numero_de_apoyos_inclinados________= #
 Numero_de_familias___(optimizacion?)___= #
 Tipo_de_problema_____estatico_o_dinam__= #

#\\\\\==============DEFINICION_DE_BARRAS==============/////
#No_material Nod_ini Nod_fin [RNod_aux] [RTipo_de_barra] [Rot_sobre_eje_prop]
#[ang_1;ang_2;ang_3]

#\\\\\==============COORDENADAS_DE_NODOS==============/////
#x y z

#\\\\\==============RESTRICCIONES==============/////
#nodo x y [z] Rx [Ry] [Rz] Presc_x Presc_y [Presc_z] Presc_Rx [Presc_Ry]
#[Presc_Rz]

#\\==============DEFINICION_DE_MATERIALES==============/////
#Modulo_Elastico Area Inerc_x [Inerc_y] [Inerc_Polar] [Modulo_de_Corte]
#Peso_Esp. Tipo_Material Nombre_Cat Num_Sec

#\\\\\==============APOYOS_ELASTICOS==============/////
#Nodo x y [z] Rx [Ry] [Rz]

#\\\\\==============APOYOS_INCLINADOS==============/////
#Nodo Inclinacion

#\\\\\==============CASOS_DE_CARGA==============/////
#Titulo
#NF_nodos NF_punt [NFpunt_y] NF_dist [NF_dist_y] [NF_triang] [NF_parabol]
#F_peso_prop
```

transversal, de lo contrario se utilizarán tres ángulos directores. Después el tipo de barra se considera como *Empotrada-Empotrada*, entonces se le coloca un número fijo 2. El siguiente valor corresponde al valor del ángulo de rotación de la sección transversal, este valor se utiliza para crear la matriz de rotación en las matrices de marcos tridimensionales. Sin embargo no se van a tomar en cuenta rotaciones de la sección transversal que no estén alineadas según los catálogos incorporados, por ello se le colocará un 0.

Posteriormente en la entrada de datos se necesitan las coordenadas de nodos. Este procedimiento es más sencillo ya que solamente hay que utilizar la información obtenida antes pero asegurando que las coordenadas se encuentran en *cm*, tal como lo planteamos antes. En cuanto a las restricciones, se utilizaron los casos más comunes en el análisis de edificaciones, *Apoyos Totalmente Restringidos - Empotrados* o *Apoyos con Desplazamientos Restringidos - Articulados*. Se decidió de esta forma ya que como se ha comentado antes, estos módulos están planeados para la fase de planeación. Esta opción la propone el usuario al utilizar el módulo ***Analysis***. Se necesita de los índices de los nodos en la base, para asignar las restricciones se utiliza una condicional, si es *Empotrado* se trazará un cubo como representación de la condición, el caso contrario para la *Articulación* se mostrará un cono.

La definición de materiales en **MECA** tiene opción de escribir las propiedades directamente, o en las últimas versiones se puede utilizar catálogos de material, ésta fue la forma seleccionada. Esto brinda rapidez al proponer secciones y materiales, además de modificarlos rápidamente si no cumplen con las especificaciones de diseño. Por esta razón se dará valores nulos a las propiedades mecánicas con excepción del peso específico, sin embargo el tipo de material se asignará conforme a los catálogos. El apartado de apoyos elásticos e inclinados queda vacío, y en casos de carga se incorporarán las fuerzas distribuidas en cada barra que se obtuvieron en el módulo ***Grid***.

Esta hoja de texto (.dat) se almacenará en una carpeta denominada *"MECA/ MECA_DYNAMO"* localizada en la carpeta *"StruturalDyn"* en el apartado de documentos. Ahora para analizar la edificación se utilizará el software **MECA**, como se ha mencionado antes. Por lo tanto se dará una breve explicación del origen, funcionamiento y características de este programa en la sección 3.7.4, antes de continuar con el procedimiento para ejecutar el análisis.

3.7.4. MECA 1.0: Software para el Análisis Matricial de Rigideces

Como se ha mencionado antes, el software de análisis matricial estructural se denomina **MECA**. Éste es un software desarrollado en la facultad de Ingeniería Civil de la Universidad de Guanajuato con la ayuda del Centro de Investigación en Matemáticas (CIMAT). El **MECA** es un paquete de cómputo que permite de forma rápida y amena resolver problemas de pequeños a grandes análisis y realizar una optimización de estructuras sujetas a diversas cargas. Este software está formado principalmente por cinco diferentes programas que interactúan y se complementan:

- *Preproceso:* Este módulo permite introducir y/o editar los datos de entrada de una estructura.
- *Proceso:* Realiza el cálculo matricial y resuelve el sistema de ecuaciones.
- *Postproceso:* Es un visualizador de resultados, utilizando los datos que retorna el módulo de *Proceso* muestra de forma amigable las fuerzas, desplazamientos u otro resultado importante.
- *Reporte:* Se genera un reporte de resultados en base al cálculo realizado.
- *Optimización:* Utiliza los datos de entrada iniciales para generar propuestas estructurales más eficientes.

Además las últimas versiones de este software tiene incorporado módulos de diseño en cuanto al material que se esté utilizando, se hace una verificación de la resistencia de cada elemento de acuerdo a normativa de acero rolado en caliente, acero rolado en frío y concreto. Es por ello que se crearon catálogos de material, los cuales el programa utiliza para generar propuestas más eficientes haciendo propuestas de secciones mediante un algoritmo de optimización.

MECA ha pasado por diversas modificaciones desde que fue creado, la primer versión tiene un entorno gráfico compatible con *Windows 95*, y publicado en 1997. Esa versión tiene cierto procedimiento de utilización, el cual denotamos a continuación:

1. Inicio de módulo “Lanzador”
2. Se utiliza un archivo de entrada , se puede crear uno nuevo o utilizar uno ya creado.
3. Utilizando el módulo de introducción de datos, es posible crear o modificar un archivo de entrada de propiedades estructurales, desde cuestiones geométricas, tipos de materiales, restricciones o cargas impuestas.

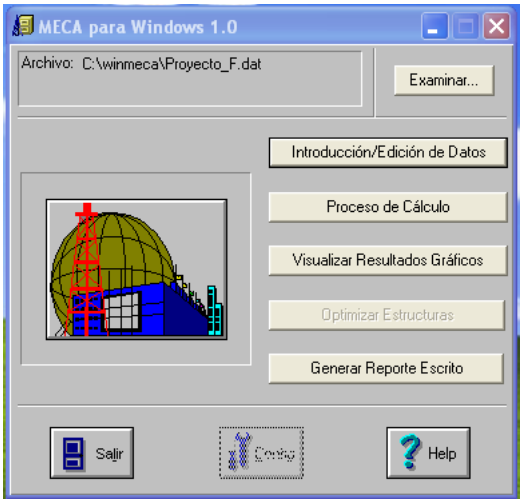


Figura 3.82: MECA 1.0. Módulo "Lanzador" del software MECA.

4. Al desear hacer el análisis existen dos opciones: *a) Proceso de Cálculo* u *b) Optimización*. Si se elige el inciso *a)*, el programa hará el calculo numérico mediante la opción de elegir el "solver" a utilizar. Si es la opción *b)* la seleccionada, entonces se procede a realizar la optimización mediante valores iniciales proporcionados.

5. Al finalizar se puede visualizar los resultados de manera gráfica mediante un entorno amigable, con la posibilidad de solicitar reportes escritos con las acciones mecánicas importantes por cada barra.

A continuación se presentan los archivos principales que el software **MECA** hace uso cuando se solicita hacer un análisis estructural:

Extensión	**Descripción**
.DAT	Archivo de datos generado por el programa de Introducción - Edición de Datos, necesario para el Proceso de Cálculo
.RES	Archivo de resultados generado por el módulo de Proceso de Cálculo, utilizado por el módulo Visualizador y Generador de Reportes.
.OPT	Archivo de resultados generado por el módulo Proceso de Optimización.
.RPT	Archivo reporte en ASCII.

Tabla 3.3: Tabla de relación de extensiones de archivos utilizados por el software **MECA**.

En cuanto al funcionamiento de usuario, el procedimiento implementado en la década de los 90's en las computadoras que se disponían era adecuado. Sin embargo hoy en día, existen muchas aplicaciones que permiten una entrada de datos más amigable como lo son los sistemas *CAD*. Por lo que la forma de realizar este proceso se vuelve un poco tediosa, sin embargo se muestran algunas imágenes de su entorno gráfico de manera ilustrativa. La imagen 3.82

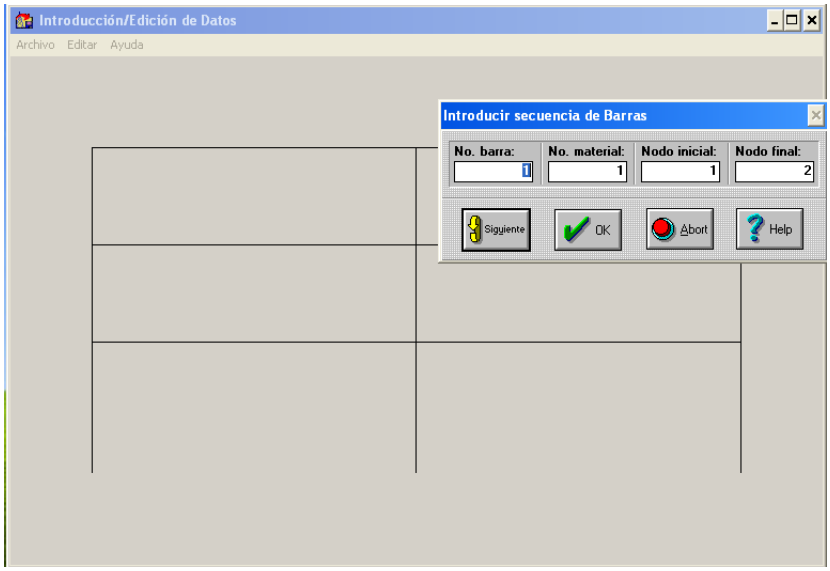


Figura 3.83: MECA 1.0. Ventana de trazo de estructura de acuerdo a la información en el módulo de "Introducción/Edición de Datos".

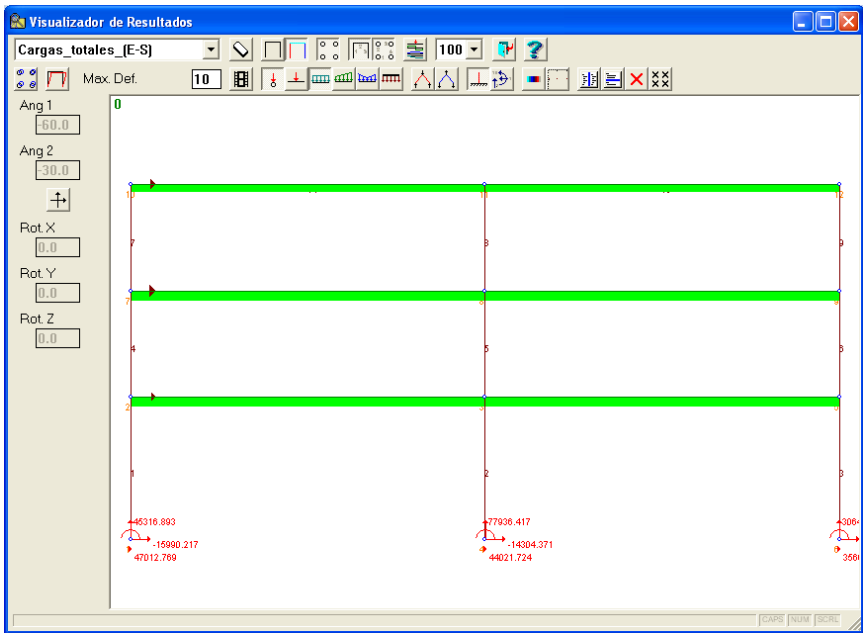


Figura 3.84: MECA 1.0. Ejemplo de solución de marco en el módulo de visualización de resultados.

corresponde a la ventana inicial del software, la cual se denomina "Lanzador", aquí se muestran las opciones principales para ejecutar los diferentes módulos. Cuando se selecciona un archivo ya creado o se decide crear uno nuevo, el siguiente paso es entrar al módulo "Introducción/Edición de Datos", el cual se compone de una ventana la cual da opciones para añadir la información de la estructura mediante ventanas emergentes. Es en la primer ventana donde se indica la volumetría de los elementos estructurales (barras, nodos, materiales, etc.) y de acuerdo a esto, van apareciendo el número de ventanas necesarias. Esta metodología es un poco tediosa, ya que los valores se van añadiendo de uno por uno, y si existen errores cometidos por el usuario, hay que volver a empezar para encontrar la ventana donde hay que realizar el cambio. Al momento de añadir la información geométrica estructural, en la ventana gráfica del módulo "Introducción/Edición de Datos" se traza la configuración estructural que fue previamente descrita mediante las coordenadas que se capturaron en las ventanas emergentes, figura 3.83.

Al tener definido la estructura a analizar, se debe hacer seleccionar el método numérico para realizar el cálculo, dependiendo a éste se podrá tener mayor eficiencia. Las computadoras de hoy en día no tienen problemas al resolver sistemas "grandes" con estos métodos, se presenta una tabla considerando que lo mencionado aquí es de propósito ilustrativo e incluso histórico.

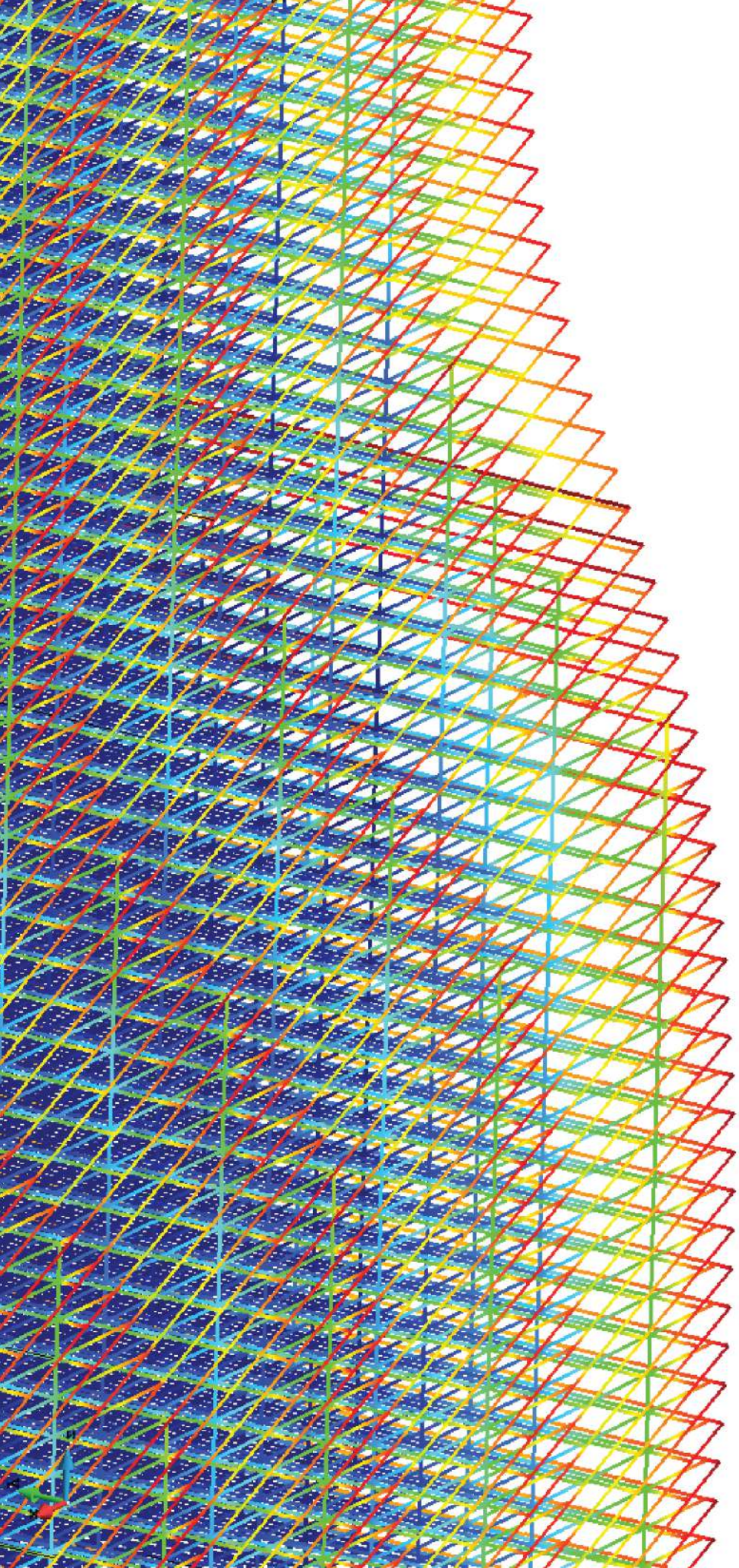
Método	**Recomendación**
Eliminación Gaussiana	Método más utilizado para el cálculo de estructuras, aunque no el más eficiente, se recomienda su uso en computadoras de memoria estándar (4 a 8 MB) y con disco duro de velocidad promedio, su velocidad de cálculo es mediana.
Factorización de Crout	Es un método de cálculo muy rápido, se recomienda su uso en la mayoría de los casos, si tiene un ordenador con suficiente memoria (8MB o más) y si la estructura a calcular es compleja.
Gradiente Conjugado	Método extremadamente lento, pero tiene la virtud de requerir una cantidad de memoria mínima para realizar el cálculo, aproximadamente 256 KB sin importar la complejidad de la estructura.

Tabla 3.4: Tabla de recomendaciones de selección de método numérico para solución de sistemas de ecuaciones por el software **MECA**, se presenta de forma que sirva como reseña histórica ya que la memoria utilizada por las computadoras de hoy es por mucho más grande.

Posteriormente al Proceso de Cálculo, se activa el "Visualizador de Resultados", el cual permite observar el comportamiento de la estructura de forma sencilla y amigable. Este módulo da la opción de observar distintos casos de carga, desplazamientos de nodos, deformación de la estructura y fuerzas en barras, figura 3.84.

Para realizar el análisis, se agregó un condicional que el usuario decide mediante una entrada *booleana* propia de la librería de **Dynamo**. Este es un botón tipo *combobox* que si el usuario determina activar, el análisis se realizará. La forma de activar el análisis es mediante la ejecución de un archivo *.exe* que se compiló utilizando el código del software **MECA** más actualizado. Ahora, tal como vemos en la tabla 3.3, cada módulo del software **MECA 1.0** necesita un tipo de dato diferente para su funcionamiento. Durante el análisis estructural ejecutado por los módulos programados en **Dynamo**, solamente nos concentraremos en dos tipos de archivo, la entrada de datos *.dat* y el archivo de resultados *.res*. Entonces este programa busca un archivo complementario denominado *coman.dat*, donde se indica el archivo de entrada (.dat) que se analizará. Entonces el primer paso es abrir ese archivo *coman.dat*, modificarlo, escribir el nombre y dirección del archivo de entrada de datos. Ahora para que el módulo ***Data_ Input_ Meca***, programado en Python, ejecute el archivo (.exe) de **MECA**, se utilizó una estrategia mediante un archivo de tipo *.bat*, que es un archivo permitido de texto que el sistema operativo reconoce y puede ejecutar. Entonces el módulo escribe un archivo (.bat) con el nombre del programa compilado (.exe) y así es posible ejecutar el análisis estructural. Se creará una carpeta en donde se guardará el archivo de entrada, el archivo de resultados entre otros que el programa de análisis arroja. Se decidió que el nombre de cada archivo tiene la hora y fecha del momento en que se realizó el análisis para poder identificar cada ejecución.

Todo el procedimiento de utilización del software **MECA 1.0** de 1997 mostrado antes en esta sección ya no es utilizado, debido a que el último sistema operativo que puede instalarlo es *Windows XP*. Es por ello que **MECA** ha evolucionado pero no de manera comercial, sino que se ha mejorado el código y se ha utilizado con propósitos de investigación y académicos. La idea ahora, es que la versión actual de **MECA** tenga una nueva forma de ingresar datos para el cálculo estructural, además de la posibilidad de visualizar resultados. Esto tiene que ver con la motivación de realizar este proyecto. Por lo que hasta antes de este proyecto, la única forma de utilizar este programa era mediante sentencias de código y utilizando archivos de texto para la transmisión de datos como el mostrado en la figura 3.81, pero ahora se puede realizar un análisis estructural y verificación de resistencia de manera muy rápida y con fácil modificación mediante la implementación de la parametrización geométrica.



Comportamiento Estructural.
Visualización de resultados utilizando GiD de una edificación con miles de barras generada con los módulos programados en este trabajo.

Capítulo 4

Resultados

Este capítulo tiene la finalidad de mostrar la funcionalidad de los modelos paramétricos en el diseño y análisis estructural. La manera en como se estructuró es principalmente en dos partes, la primera corresponde a las mediciones cuantitativas y cualitativas que se describieron en el capítulo 2 denominado "Parametrización Geométrica" y más específicamente en la sección 2.2.2 "Eficiencia de los Modelos Paramétricos". La segunda parte de este capítulo corresponde a la medición de la latencia de acuerdo a la estructuración y análisis de cuatro modelos sólidos paramétricos que denotan edificaciones, en donde se prueban distintas configuraciones estructurales para hacer hincapié en el rápido proceso de modificación mediante el uso de parámetros.

4.1. Medición de la Eficiencia y Flexibilidad del Modelo Paramétrico

A continuación se presentan las mediciones cuantitativas y cualitativas propuestas por Davis (9), que como ya se mencionó antes, aún cuando no existe una medición estandarizada para modelos paramétricos, se realizó una revisión de estos conceptos para tratar de relacionarlos con la eficiencia y flexibilidad de los módulos programados en este trabajo.

Medidas Cuantitativas

- **Tiempo de construcción.** Esta es una característica importante para los modelos paramétricos en general, ya que se deben proponer funciones y realizar las debidas conexiones. Sin embargo en este trabajo los módulos son muy pocos, por ello es muy fácil construir el modelo y el tiempo requerido es muy corto, se toma un par de minutos realizar la construcción a un usuario estándar.
- **Tiempo de modificación.** Este concepto se puede referir desde hacer un cambio a un parámetro o hacer cambios en las funciones. Al igual que

en la medición anterior, se comentó que los módulos son muy sencillos de conectar, y ya no es necesario hacer modificaciones en las funciones que los integran. En cuanto al cambio en los parámetros, solo basta con cambiar valores numéricos las entradas o inputs, los cuales son un número pequeño, por tanto el tiempo de modificación se considera muy corto, solo de algunos minutos para un usuario familiarizado con el modelador.

- **Latencia.** Como se mencionó antes, este corresponde al tiempo de espera al realizar un cambio en algún parámetro. Ya que se hicieron varias pruebas de modelos, los resultados de la latencia se muestran en la segunda parte de este capítulo. Sin embargo se puede decir que en general, la estructuración, análisis y diseño de una edificación es considerablemente más rápida mediante el uso de estos módulos programados, que utilizando un procedimiento tradicional. Los datos que respaldan esta aseveración se mostrarán en forma gráfica y mediante tablas posteriormente.

- **Dimensionalidad.** La dimensionalidad es el número total de parámetros de un modelo, en este caso tomaremos los parámetros mínimos para realizar la estructuración, análisis y verificación del diseño de una edificación. Son solo 16 parámetros necesarios para realizar todo este proceso, los cuales se nombran a continuación:

  1. Sólido.
  2. Altura del sólido.
  3. Número de niveles.
  4. Tamaño en X de la retícula de estructuración.
  5. Tamaño en Y de la retícula de estructuración.
  6. Número de pórticos en dirección X.
  7. Número de pórticos en dirección Y.
  8. Propiedad de sección de material para vigas.
  9. Piso inicial de utilización de sección para vigas.
  10. Piso final de utilización de sección para vigas.
  11. Propiedad de sección de material para columnas.
  12. Piso inicial de utilización de sección para columnas.
  13. Piso final de utilización de sección para columnas.
  14. Magnitud de fuerza por unidad de área (Kg/m^2) aplicadas en losas.
  15. Booleano de restricción en apoyos (empotrados/artículados).
  16. Booleano para ejecutar análisis (verdadero/falso).

Fase	Módulo	LOC - DAG	LOC - Python
Materiales	Concreto/Acero	1	70
Estructuración	Floors	1	142
	Grid	1	492
	Structure	1	1115
Cargas	Slab Loads	1	971
Análisis	Data Input MECA	1	328
	Post Pro	1	178
	Total	7	3296

Tabla 4.6: Líneas de código LOC. Tabla que muestra la comparación entre la medición de líneas de código para modelos paramétricos usando redes DAG y el número de líneas programadas con lenguaje Python para cada módulo propuesto en este trabajo.

- **Tamaño.** El tamaño de un programa predice si existe la posibilidad de tener errores, ya que mientras más grande es el código, es mas complejo. Utilizando esta medición para modelos paramétricos, tenemos que un módulo o nodo de la red DAG equivale a una línea de código, por tanto se contó el número de módulos pero además también las líneas de código que contiene cada uno.
- **Complejidad ciclomática.** Tal como se menciona en la sección 2.2.2, la complejidad ciclomática relaciona el número de nodos, conexiones, parámetros de entrada y salida para regresar un valor que mientras más pequeño sea es mejor, debido a que es más probable que tenga menos errores. Esta medición se adaptó para modelos paramétricos que utilizan un gran número de redes mediante la fórmula de Henderson-Seller y Tegarden, sin embargo también la utilizamos para tener una referencia en cuanto a otros modelos. Entonces a continuación se muestra la complejidad adquirida para un modelo utilizando los módulos programados en este trabajo, se midieron únicamente los nodos que son necesarios para la estructuración, cargas y análisis, siendo éste último el resultado esperado.

$$CC(G) = (e + (i - 1) + (u - 1)) - n + 2$$

Donde:

- G: La gráfica.
- e: Número de conectores. Siendo un solo conector las relaciones paralelas.
- n: Número de nodos. Sin contar nodos no funcionales (como los de texto).
- i: Número de entradas (inputs).
- u: Número de salidas (outputs).

$$CC(G) = (21 + (16 - 1) + (1 - 1)) - 7 + 2 = \mathbf{31}$$

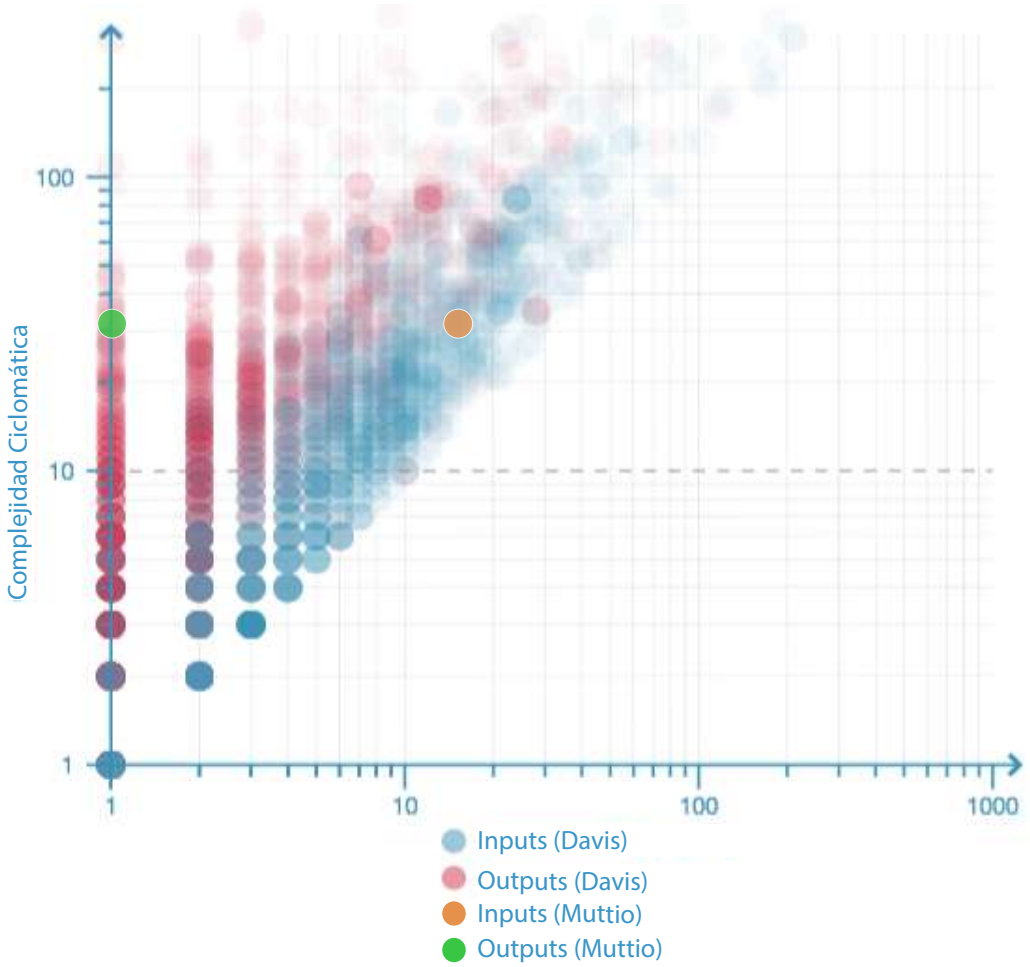


Figura 4.1: Complejidad Ciclomática. Gráfica obtenida de la revisión de 2002 modelos paramétricos por Davis ((9)), donde agregamos dos mediciones de acuerdo al resultado obtenido por los módulos programados en este trabajo.

En este caso, se obtuvo una complejidad de 31, según la teoría revisada un software (de cualquier tipo) debe tener una magnitud no mayor a 10. Sin embargo, en el trabajo de Davis (9), se hicieron 2002 pruebas de modelos paramétricos mediante programación visual y encontró que este límite de 10 es imposible en este campo. Por ello presenta una gráfica que compara tanto la complejidad ciclomática de acuerdo al número de "inputs" como con el número de "outputs". Observando los puntos añadidos en color naranja (inputs) y verde (outputs) correspondientes a la medición de este trabajo, se puede observar que se encuentra en una buena posición, justo en el promedio de los modelos probados por Davis. Mientras un modelo tenga un valor más pequeño, éste se considerará más eficiente.

Medidas Cualitativas

- **Exactitud.** Referida a la capacidad del programa para realizar la tarea que se le asigno. De acuerdo a los experimentos realizados, se ha cumplido este objetivo en la mayoría de los casos de estudio. Tal como se comentó en el capítulo 3, existen algunas funciones propias de **Dynamo** que presentan cierta inestabilidad. Además hay que considerar que quizá existan más casos no probados, que tengan problemas en la estructuración y análisis. Obtener una exactitud en la mayoría de los modelos es un trabajo arduo que podrá seguir siendo investigado e implementado en un trabajo futuro.

- **Robustez.** Referida al comportamiento debido al uso incorrecto del software. Esta medición es muy importante a la hora de lanzar un producto que será utilizado por miles de personas, ya que el programa debe tener cierta resiliencia a problemas originados por el desconocimiento del usuario del funcionamiento correcto del programa. Sin embargo, debido a que este es un proyecto académico, se utilizó el modelador paramétrico correctamente en todo momento.

- **Reutilización.** En cuanto a la estructura del código, se puede decir que es altamente reutilizable, debido a que muchas funciones pueden utilizarse para diferentes procesos dentro de la misma aplicación. Ahora, también es muy reutilizable la aplicación en su totalidad para diferentes cuestiones, como por ejemplo otro tipo de análisis en edificios. Debido a su disposición modular, un usuario puede tomar solamente algunos módulos que le parezcan necesarios y conectarlos a otro evaluador para tener una nueva aplicación.

- **Eficiencia.** Algunos módulos fueron optimizados mediante programación en paralelo para reducir la latencia que se tenía, lo cual proporciona una mayor eficiencia de la aplicación. Sin embargo, no es posible hacer

una optimización tan detallada porque en este caso, esta medición entraría en conflicto con la anterior. Mientras más especializado se vuelve un programa, menos reutilización le podemos dar. Por eso se tuvo un punto medio entre estas dos características.

- **Portabilidad**: Debido a que estamos trabajando dentro un ambiente de modelado paramétrico de un software comercial, la única transferencia que podemos realizar es a través de software compatible o de la misma familia de productos. Aunque el código está programado en Python, y esto puede ayudar a migrar a otro sistema para dar mayor portabilidad.

- **Facilidad de uso.** Esta es una de las características principales de esta aplicación, porque es muy sencillo proponer una configuración estructural de una edificación, asignar propiedades geométricas y mecánicas de las secciones transversales, imponer cargas, hacer el análisis y verificar resistencias. Incluso no hay que ser un experto en el uso del modelador paramétrico para poder ejecutar esta aplicación.

- **Funcionalidad.** Aunque se tiene una gran cantidad de modelos estructurales que se pueden analizar, siempre existirán limitantes o simplificaciones que no nos permiten tener la flexibilidad que deseamos. Dichas restricciones se mencionan en el capítulo 3. Aún así, se tiene una aplicación que facilita por mucho la posibilidad de proponer muchas soluciones estructurales de manera íntegra y muy fácil.

En las siguientes páginas se mostrarán algunos resultados gráficos de la utilización de los módulos programados. Se utilizaron cuatro modelos experimentales, los cuales fueron sometidos al proceso de estructuración, distribución de cargas, análisis y verificación de resistencia. Los primeros modelos corresponden a componentes utilizados en la edificación que se utilizó como principal motivación para realizar este trabajo, pero que por si solos pueden constituir una edificación. Por lo tanto el primer modelo es un prisma rectangular, el segundo es un elipse, el tercero es un sólido compuesto por tres polígonos (inferior, intermedio, superior) y el cuarto es el modelo de la *"Torre Mayor"*.

Por cada sólido se generaron 8 diferentes modelos, los primeros cuatro tienen las mismas dimensiones pero se prueban distintas configuraciones estructurales. Por el contrario, los últimos cuatro si tienen variación en cuanto a las dimensiones del sólido y se trata de conservar la misma configuración estructural. Se hicieron mediciones de latencia por cada modelo, por lo que se tiene una gráfica con el eje vertical en escala logarítmica correspondiente a los tiempos de actualización o latencia por cada sólido, donde se puede ver el comportamiento del algoritmo bajo distintas solicitaciones. En este caso se puede observar que mientras más barras, la latencia va aumentando con una cierta proporción, pero llega el punto en el que es necesario almacenar una cantidad

de datos tan grande que la curva se asemeja a una función exponencial.

Posteriormente se muestra una tabla con cada parámetro necesario para generar la estructura y ejecutar el análisis de cada modelo, por lo que si es necesario corroborar los resultados es de esta manera que se puede ejecutar la aplicación. también muestra el claro máximo que se puede presentar en la estructuración para tener una referencia lógica de la propuesta estructural, y además al final se muestra una pequeña sección con el tiempo de ejecución de cada módulo. En las siguientes hojas se muestra una matriz de imágenes de las edificaciones, la primera corresponde al sólido, posteriormente se observa el *grid* o retícula utilizada, después se muestra la estructura, la distribución de cargas gravitacionales, y finalmente la eficiencia de cada barra mediante la verificación de resistencia.

Los modelos que se generaron en este capítulo fueron posibles debido a la utilización de una computadora que fue brindada por el ***Centro Estatal de Supercómputo de Guanajuato - CIMAT*** para la realización de este trabajo. Este centro permite la configuración combinada de múltiples unidades de cómputo para ofrecer una alta capacidad de procesamiento simultáneo. El supercómputo es una estrategia que incluye metodologías numéricas matemáticas con las cuales se intenta resolver problemas complejos de forma más rápida y sencilla.

Al finalizar se presenta una comparación de tiempo entre el uso del equipo de supercómputo y una computadora personal sencilla (laptop) para conocer el impacto del recurso tecnológico entre los dos equipos.

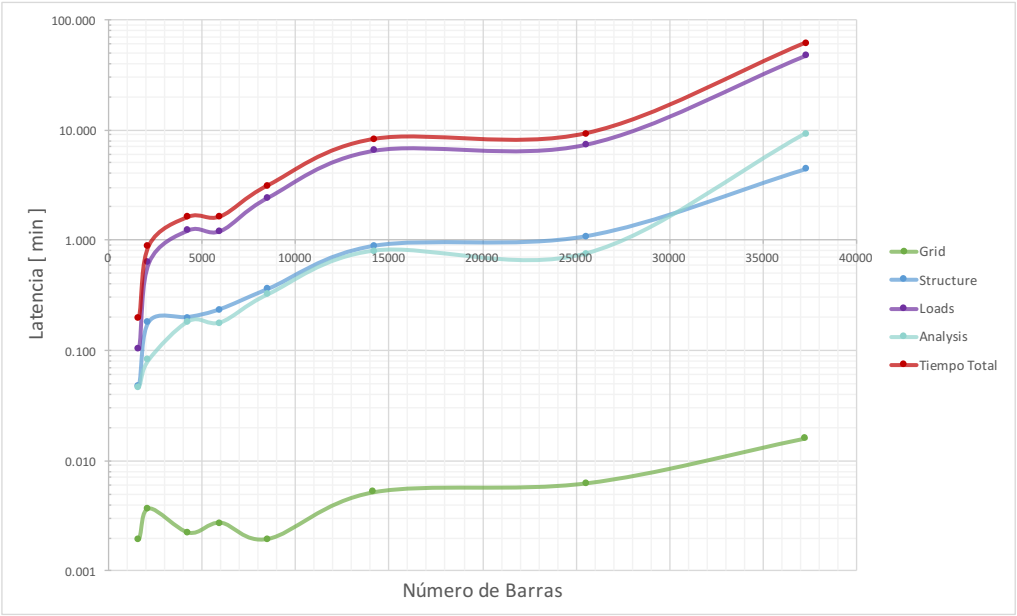
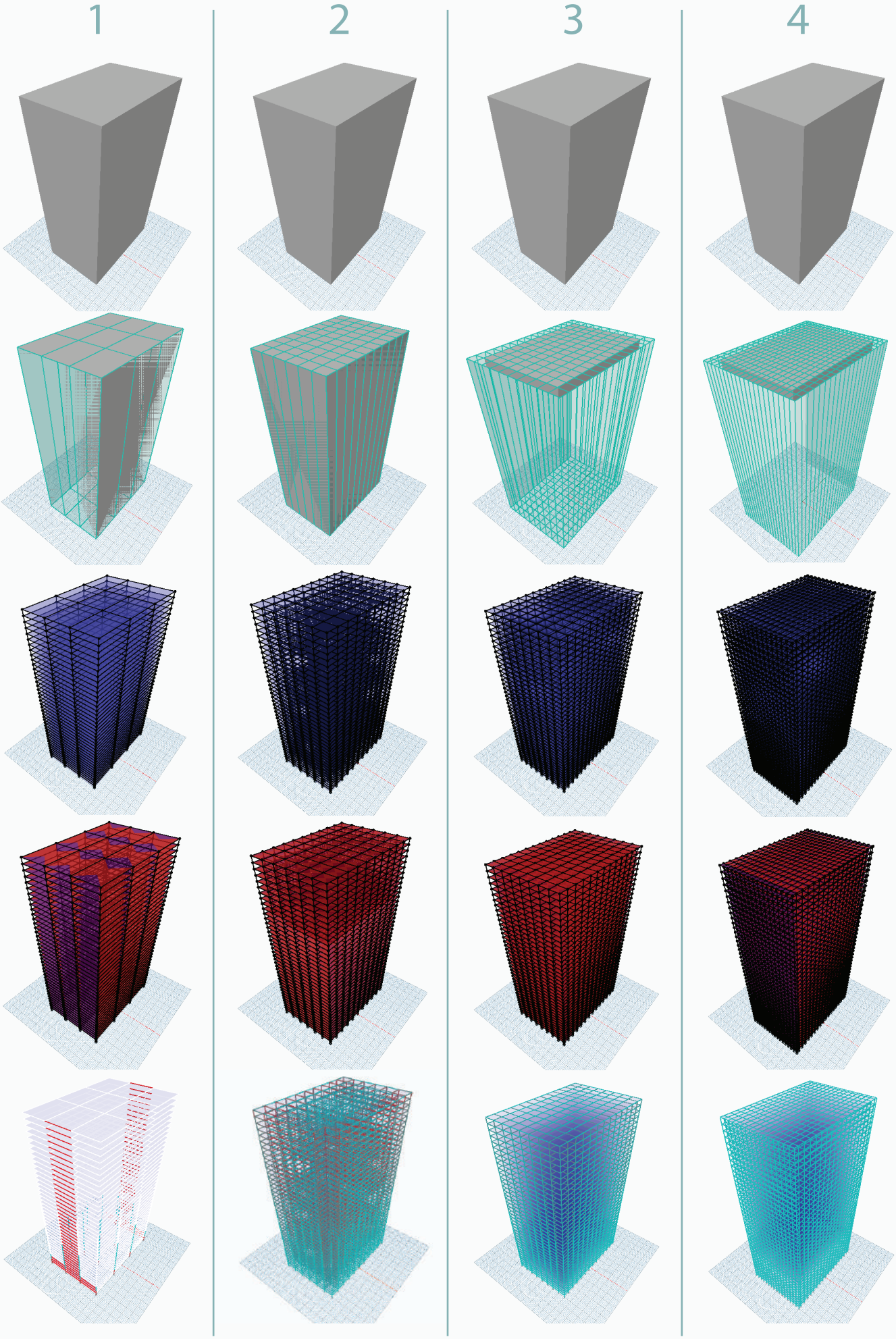
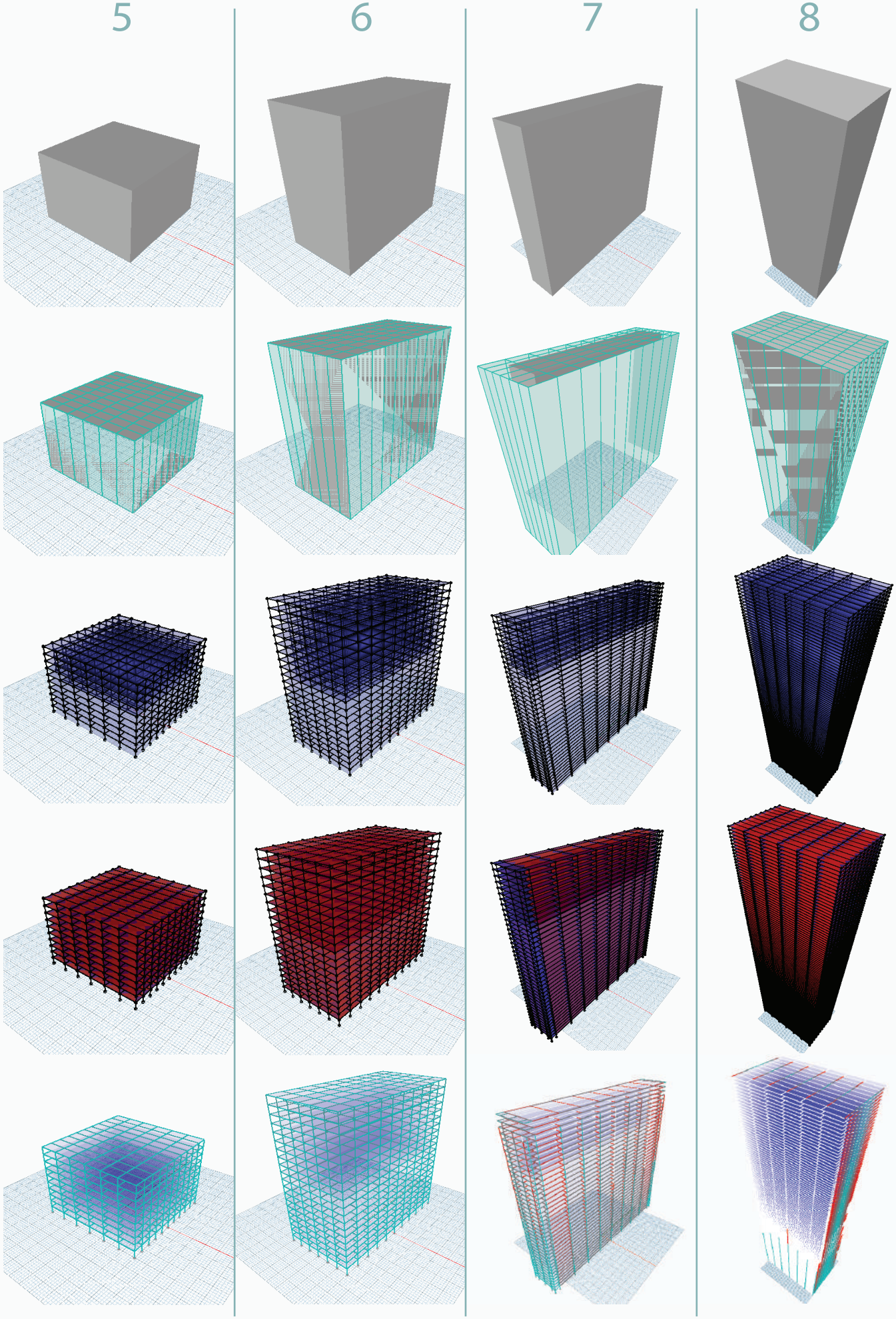


Figura 4.2: Latencia Edificio Rectangular. Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "Rec Building", se observa que para modelos menores a 25 000 barras se necesita menos de 10 min para el proceso completo. Sin embargo para el modelo #4 de 37320 barras, el tiempo necesario es seis veces mayor, por lo que se presenta un comportamiento cuasi-exponencial.

Rec Building	Pruebas en Estructuración				Pruebas en Sólido			
Solid Parameters	1	2	3	4	5	6	7	8
X-Dim [m]	40	40	40	40	40	30	20	100
Y-Dim [m]	60	60	60	60	40	60	120	60
Height [m]	120	120	120	120	30	60	120	360
Structure Parameters	1	2	3	4	5	6	7	8
NFloors	40	40	40	40	10	20	40	120
GridSize_X [m]	40	40	50	50	40	30	40	100
GridSize_Y [m]	60	60	70	70	40	60	140	60
GridSep_X [m]	3	6	10	20	6	6	6	6
GridSep_Y [m]	3	10	15	20	10	10	10	10
Claro en X max [m]	13.33	6.67	5.00	2.50	6.67	5.00	6.67	16.67
Claro en Y max [m]	20.00	6.00	4.67	3.50	4.00	6.00	14.00	6.00
Material Parameters	1	2	3	4	5	6	7	8
Vigas Concreto								
N Section	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8
First Floor	1	1	1	1	1	1	1	1
Final Floor	40	40	40	40	40	40	40	40
Columnas Concreto 1								
N Section	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6
First Floor	1	1	1	1	1	1	1	1
Final Floor	20	20	20	20	20	20	20	20
Columnas Concreto 2								
N Section	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8
First Floor	20	20	20	20	20	20	20	20
Final Floor	40	40	40	40	40	40	40	40
Slab Loads	1	2	3	4	5	6	7	8
live_load [Kg/m2]	100	100	100	100	100	100	100	100
dead_load [Kg/m2]	240	240	240	240	240	240	240	240
Analysis	1	2	3	4	5	6	7	8
Restricted Supports	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Latencia	1	2	3	4	5	6	7	8
Barras	1600	8520	14200	37320	2130	4260	5960	25560
Nodos	656	3157	5166	13243	847	1617	2255	9317
Ecuaciones	3936	18942	30996	79458	5082	9702	13530	55902
Tiempo Grid [min]	0.00193	0.00193	0.00517	0.01602	0.00367	0.00220	0.00272	0.00622
Tiempo Structure [min]	0.04780	0.36057	0.88753	4.42685	0.18180	0.19952	0.23605	1.08530
Tiempo Cargas [min]	0.10302	2.44447	6.50583	47.40018	0.62803	1.23418	1.21452	7.38552
Tiempo Análisis [min]	0.04685	0.32215	0.79785	9.25798	0.08242	0.18348	0.17842	0.75352
Total [min]	0.19960	3.12912	8.19638	61.10103	0.89592	1.61938	1.63170	9.23055

Tabla 4.7: Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de un prisma rectangular. Se incluyen los tiempos de ejecución o latencia por cada modelo.


1
2
3
4


5
6
7
8

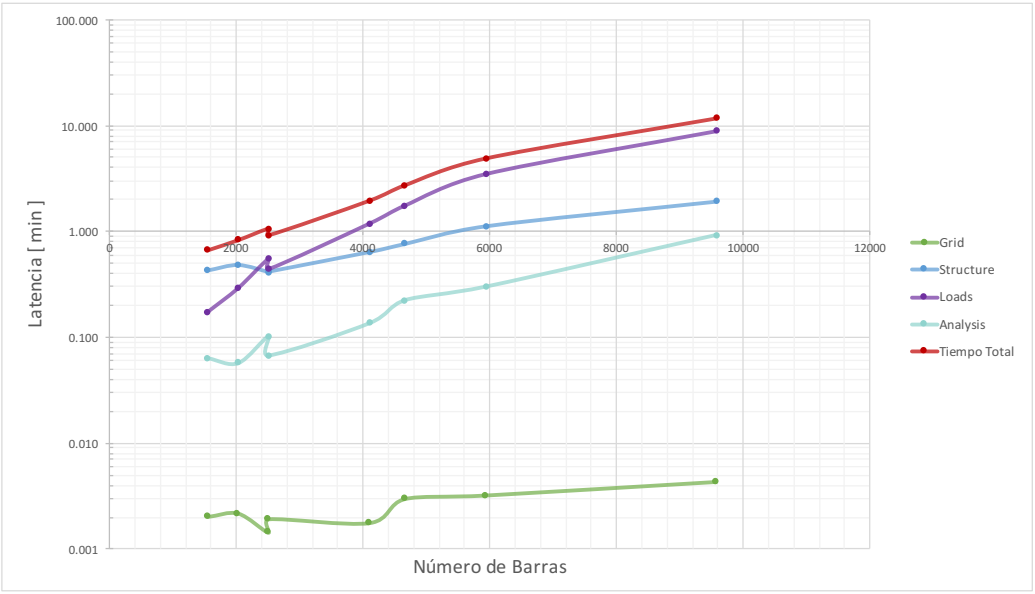
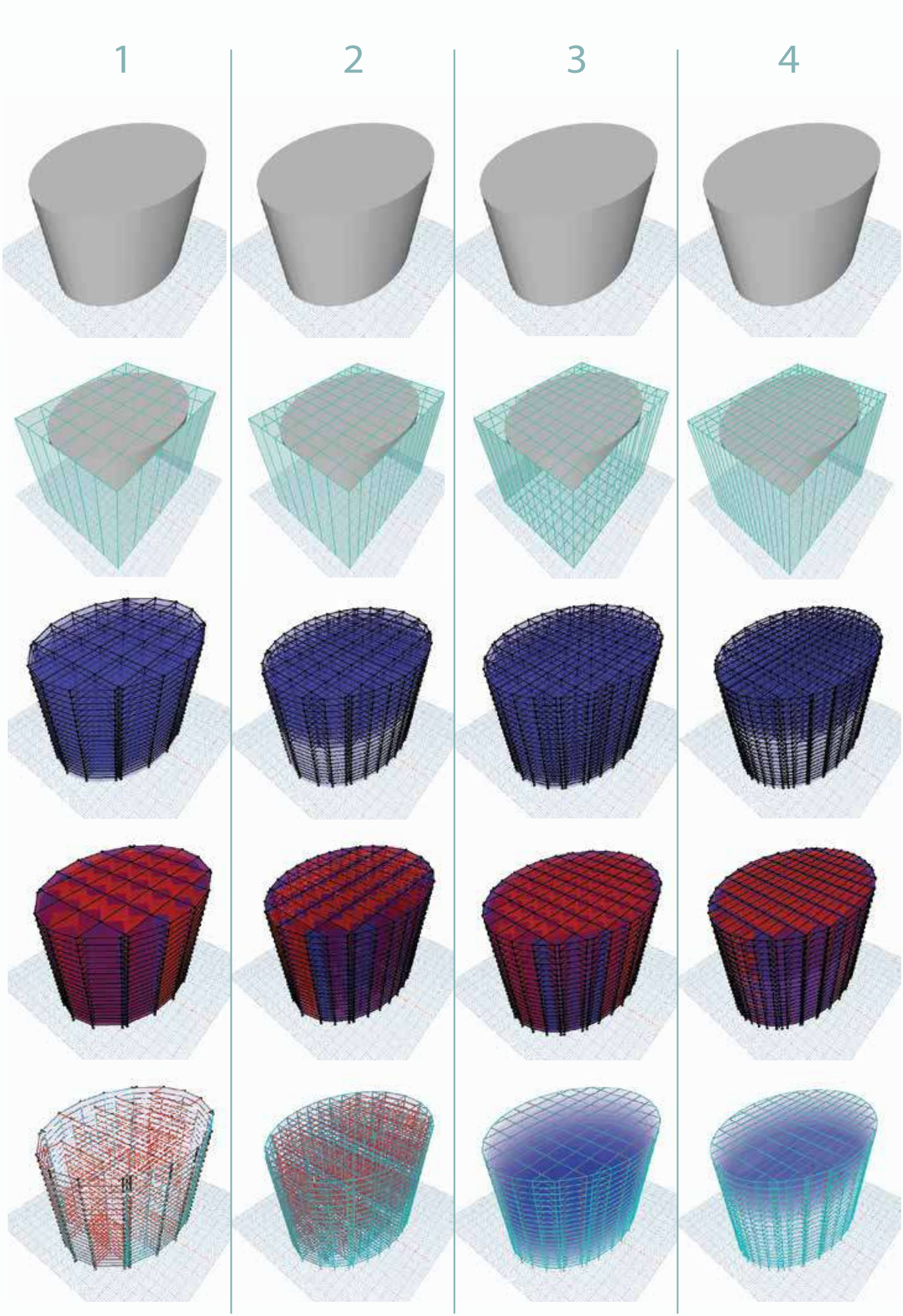
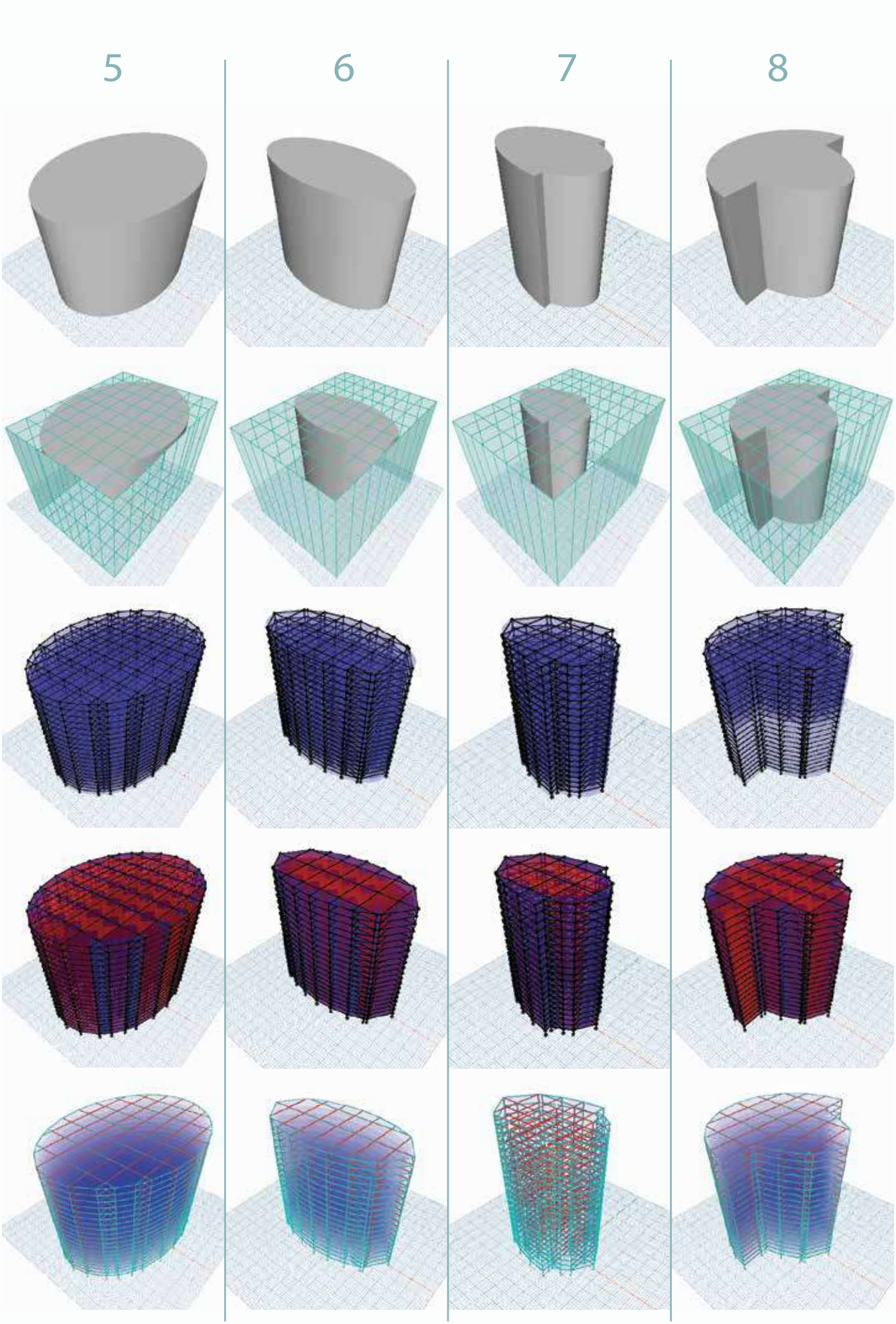


Figura 4.3: Latencia Edificio Elipsoidal. Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "Ellipse". En este caso se ejecutaron modelos más pequeños, por lo que se observa un buen comportamiento, donde mientras crece el número de barras también lo hace el tiempo de forma regular. Aquí vemos la mayoría de los modelos están por debajo de los dos minutos, lo cual es algo muy bueno ya que son edificios de tamaño regular que se construyen de manera más común que rascacielos.

Ellipse	Pruebas en Estructuración				Pruebas en Sólido			
Solid Parameters	1	2	3	4	5	6	7	8
Height [m]	60	60	60	60	60	60	60	60
X-Dim [m]	30	30	30	30	30	30	30	40
Y-Dim [m]	40	40	40	40	40	15	15	20
Offset 1 [m]	1	1	1	1	1	1	18	18
X-Dim 2 [m]	30	30	30	30	30	30	30	30
Y-Dim 2 [m]	40	40	40	40	40	40	15	30
Offset 2 [m]	-1	-1	-1	-1	-1	-1	-1	-1
Structure Parameters	1	2	3	4	5	6	7	8
NFloors	20	20	20	20	20	20	20	20
GridSize_X [m]	60	60	60	60	60	60	60	60
GridSize_Y [m]	80	80	80	80	80	80	80	80
GridSep_X [m]	6	6	10	16	8	8	10	16
GridSep_Y [m]	6	12	10	10	8	8	10	10
Claro en X max [m]	10.00	10.00	6.00	3.75	7.50	7.50	6.00	3.75
Claro en Y max [m]	13.33	6.67	8.00	8.00	10.00	10.00	8.00	8.00
Material Parameters	1	2	3	4	5	6	7	8
Vigas Concreto								
N Section	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8
First Floor	1	1	1	1	1	1	1	1
Final Floor	20	20	20	20	20	20	20	20
Columnas Concreto 1								
N Section	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6
First Floor	1	1	1	1	1	1	1	1
Final Floor	10	10	10	10	10	10	10	10
Columnas Concreto 2								
N Section	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8
First Floor	10	10	10	10	10	10	10	10
Final Floor	20	20	20	20	20	20	20	20
Slab Loads	1	2	3	4	5	6	7	8
live_load [Kg/m2]	100	100	100	100	100	100	100	100
dead_load [Kg/m2]	240	240	240	240	240	240	240	240
Analysis	1	2	3	4	5	6	7	8
Restricted Supports	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Latencia	1	2	3	4	5	6	7	8
Barras	2500	4660	5940	9580	4100	2020	1540	2500
Nodos	945	1743	2205	3507	1533	777	609	966
Ecuaciones	5670	10458	13230	21042	9198	4662	3654	5796
Tiempo Grid [min]	0.0014	0.0030	0.0032	0.0043	0.0018	0.0022	0.0020	0.0019
Tiempo Structure [min]	0.4121	0.7636	1.1245	1.9296	0.6399	0.4810	0.4271	0.4158
Tiempo Cargas [min]	0.5476	1.7497	3.4782	8.8449	1.1822	0.2889	0.1720	0.4371
Tiempo Análisis [min]	0.1022	0.2238	0.2999	0.9149	0.1362	0.0573	0.0636	0.0676
Total [min]	1.0633	2.7400	4.9058	11.6936	1.9601	0.8292	0.6647	0.9223

Tabla 4.8: Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de un prisma elipsoidal. Se incluyen los tiempos de ejecución o latencia por cada modelo.


1
2
3
4


5
6
7
8

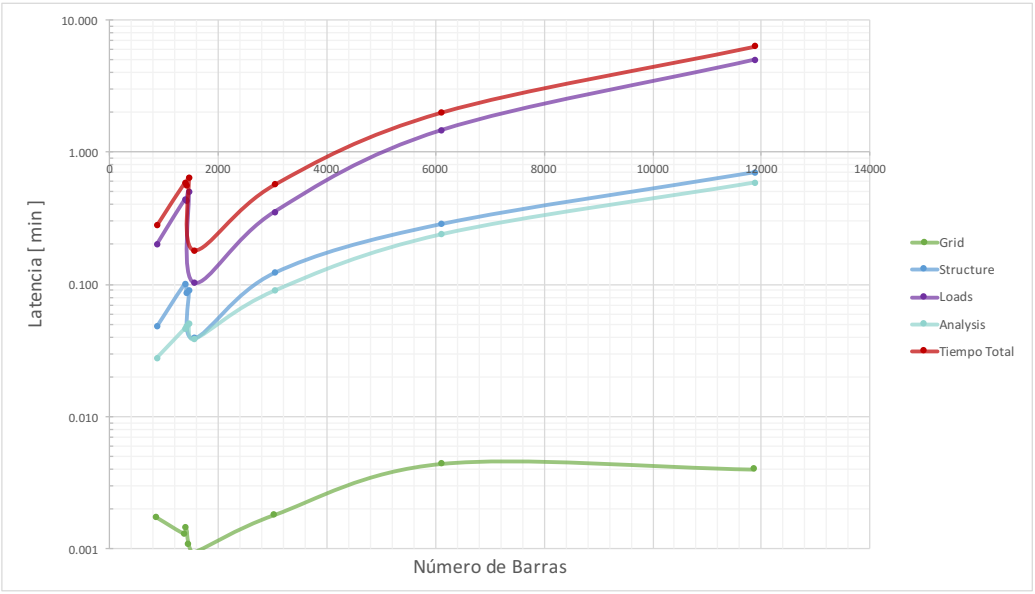
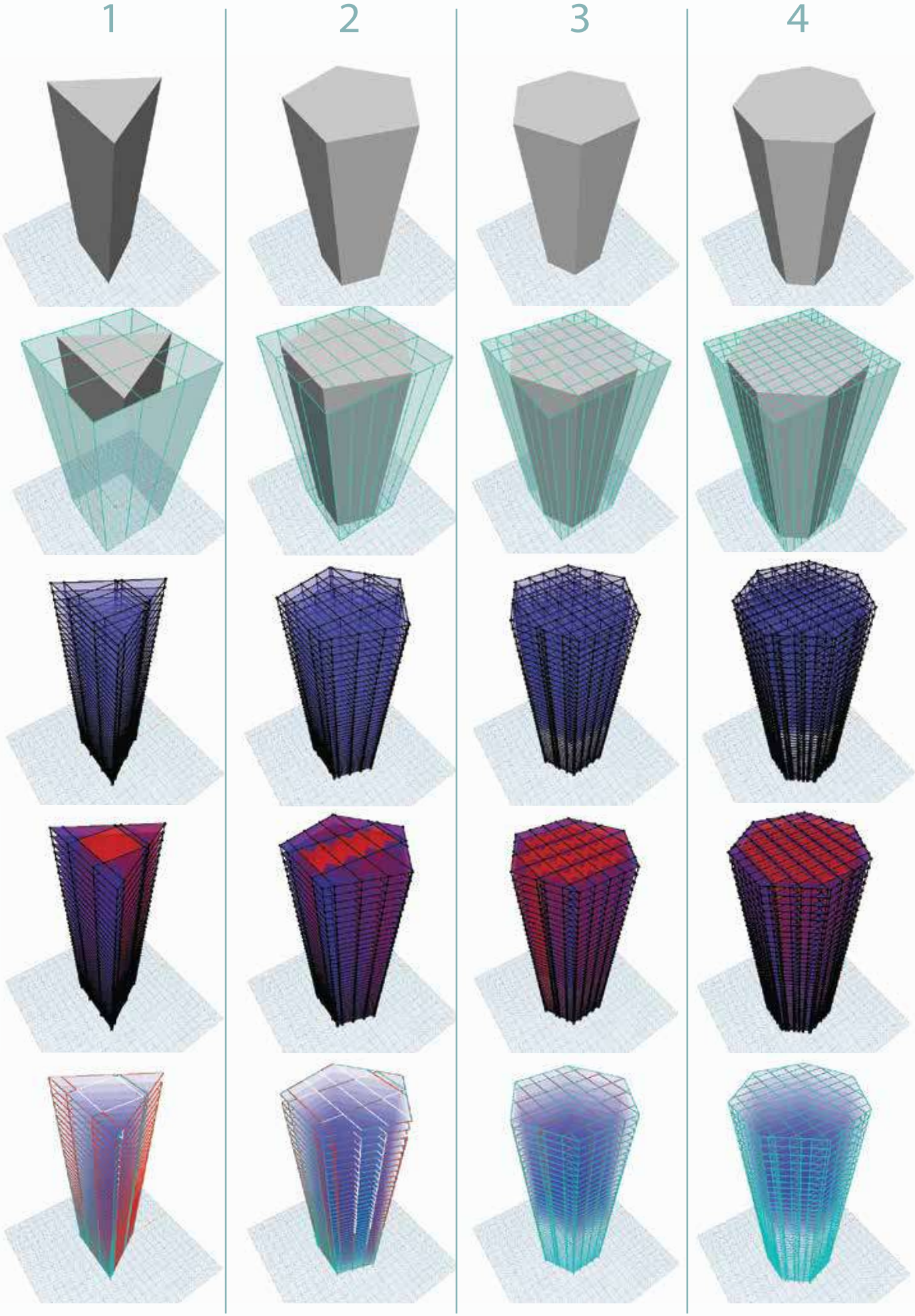
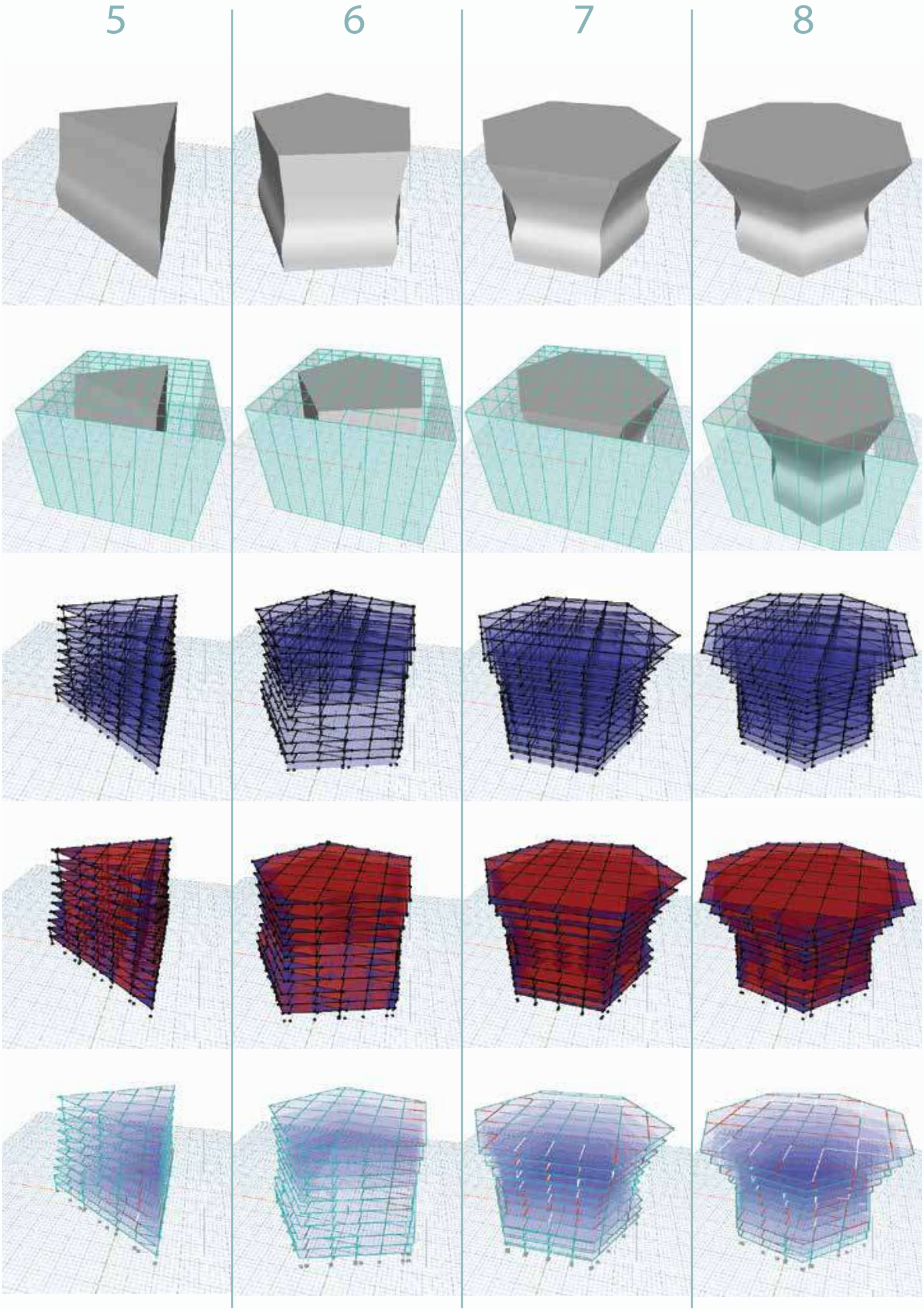


Figura 4.4: Latencia Edificio Elipsoidal. Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación con el sólido "3Pol Building". Este es un caso interesante, primero porque los modelos son pequeños, se trató de ejecutar análisis de edificaciones que podemos encontrar casi en cualquier ciudad, con no más de 12 000 barras. Por tanto los tiempos de ejecución son menores a un minuto en su mayoría. Una particularidad de esta gráfica es que al inicio de las curvas se tiene un crecimiento de latencia y luego disminución. Esto ocurre debido a que inicialmente el algoritmo no tiene activada la función de paralelización por las razones que se dieron en el capítulo correspondiente, es por ello que al activarse automáticamente, se tiene una reducción importante de tiempo en el cuarto punto mostrado.

3Pol Building	Pruebas en Estructuración				Pruebas en Sólido			
Solid Parameters	1	2	3	4	5	6	7	8
Sides	3	5	6	7	3	5	6	7
Radius 1 [m]	20	20	20	20	20	20	20	20
Radius 2 [m]	20	20	20	20	18	18	18	18
Radius 3 [m]	20	20	20	20	16	18	22	24
Height [m]	120	120	120	120	30	30	30	30
Structure Parameters	1	2	3	4	5	6	7	8
NFloors	40	40	40	40	10	10	10	10
GridSize_X [m]	40	40	40	40	40	40	45	48
GridSize_Y [m]	40	40	40	40	40	40	45	48
GridSep_X [m]	3	6	8	12	8	8	8	8
GridSep_Y [m]	3	3	8	12	8	8	8	8
Claro en X max [m]	13.33	6.67	5.00	3.33	5.00	5.00	5.63	6.00
Claro en Y max [m]	13.33	13.33	5.00	3.33	5.00	5.00	5.63	6.00
Material Parameters	1	2	3	4	5	6	7	8
Vigas Concreto								
N Section	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8
First Floor	1	1	1	1	1	1	1	1
Final Floor	40	40	40	40	10	10	10	10
Columnas Concreto 1								
N Section	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6
First Floor	1	1	1	1	1	1	1	1
Final Floor	20	20	20	20	5	5	5	5
Columnas Concreto 2								
N Section	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8
First Floor	20	20	20	20	5	5	5	5
Final Floor	40	40	40	40	10	10	10	10
Slab Loads	1	2	3	4	5	6	7	8
live_load [Kg/m2]	100	100	100	100	100	100	100	100
dead_load [Kg/m2]	240	240	240	240	240	240	240	240
Analysis	1	2	3	4	5	6	7	8
Restricted Supports	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Latencia	1	2	3	4	5	6	7	8
Barras	1560	3040	6120	11880	866	1456	1408	1384
Nodos	615	1189	2255	4387	461	719	699	705
Ecuaciones	3690	7134	13530	26322	2766	4314	4194	4230
Tiempo Grid [min]	0.0010	0.0018	0.0044	0.0040	0.0017	0.0011	0.0014	0.0013
Tiempo Structure [min]	0.0391	0.1218	0.2844	0.6967	0.0479	0.0903	0.0865	0.1003
Tiempo Cargas [min]	0.1019	0.3522	1.4579	4.9499	0.1991	0.4977	0.4230	0.4379
Tiempo Análisis [min]	0.0383	0.0891	0.2380	0.5853	0.0276	0.0507	0.0477	0.0462
Total [min]	0.1803	0.5649	1.9848	6.2359	0.2763	0.6399	0.5587	0.5857

Tabla 4.9: Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de varios prismas generados por el módulo "3Pol". Se incluyen los tiempos de ejecución o latencia por cada modelo.


1
2
3
4


5
6
7
8

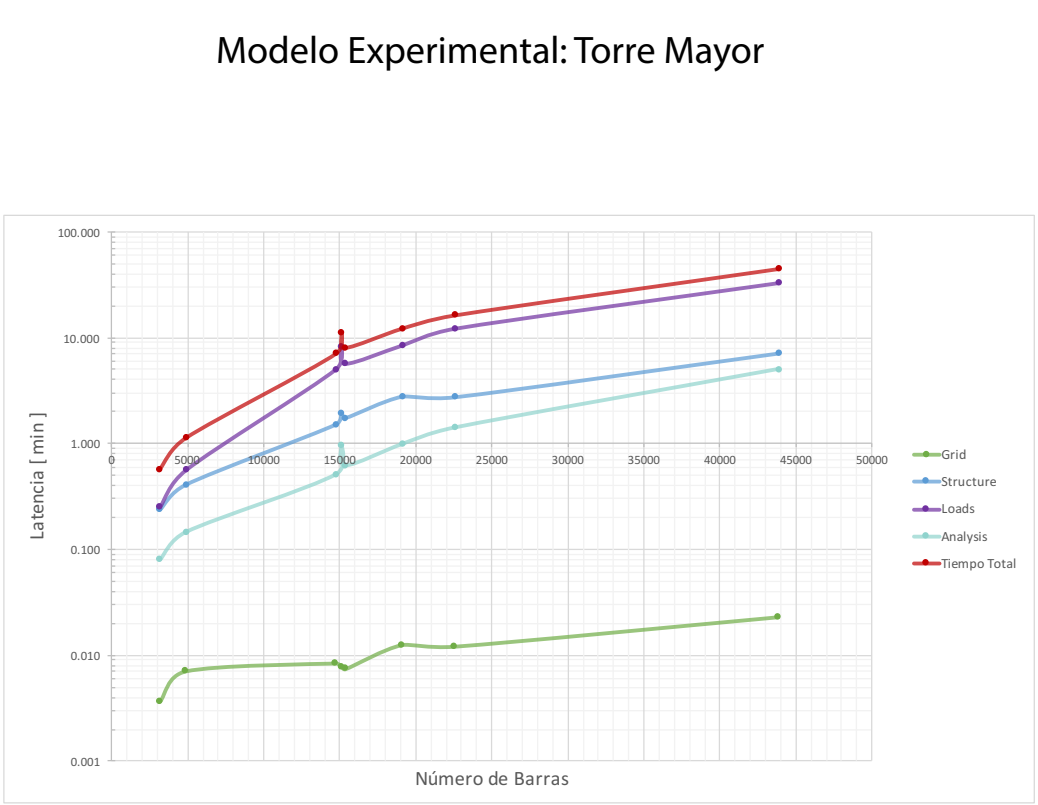
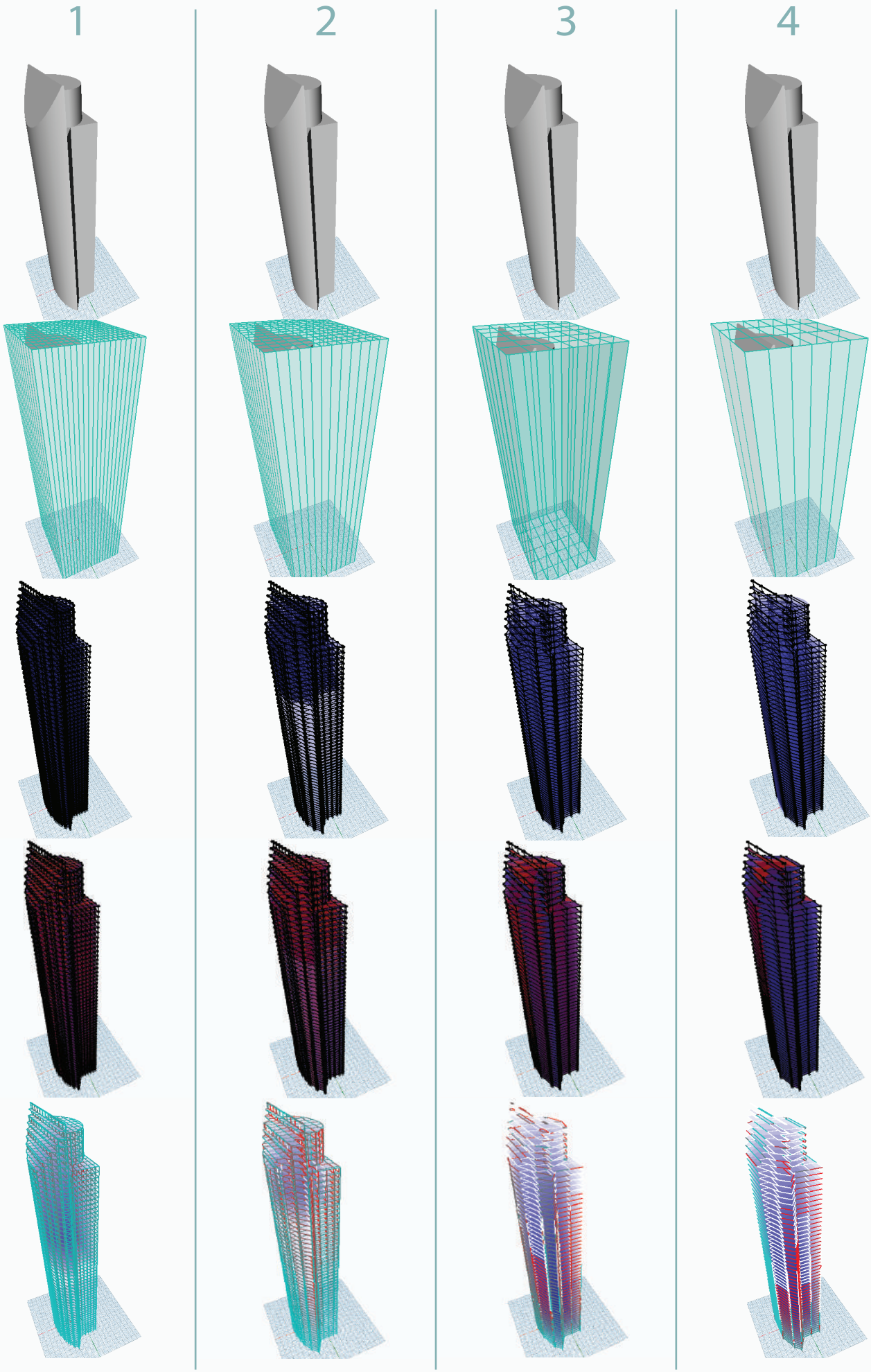
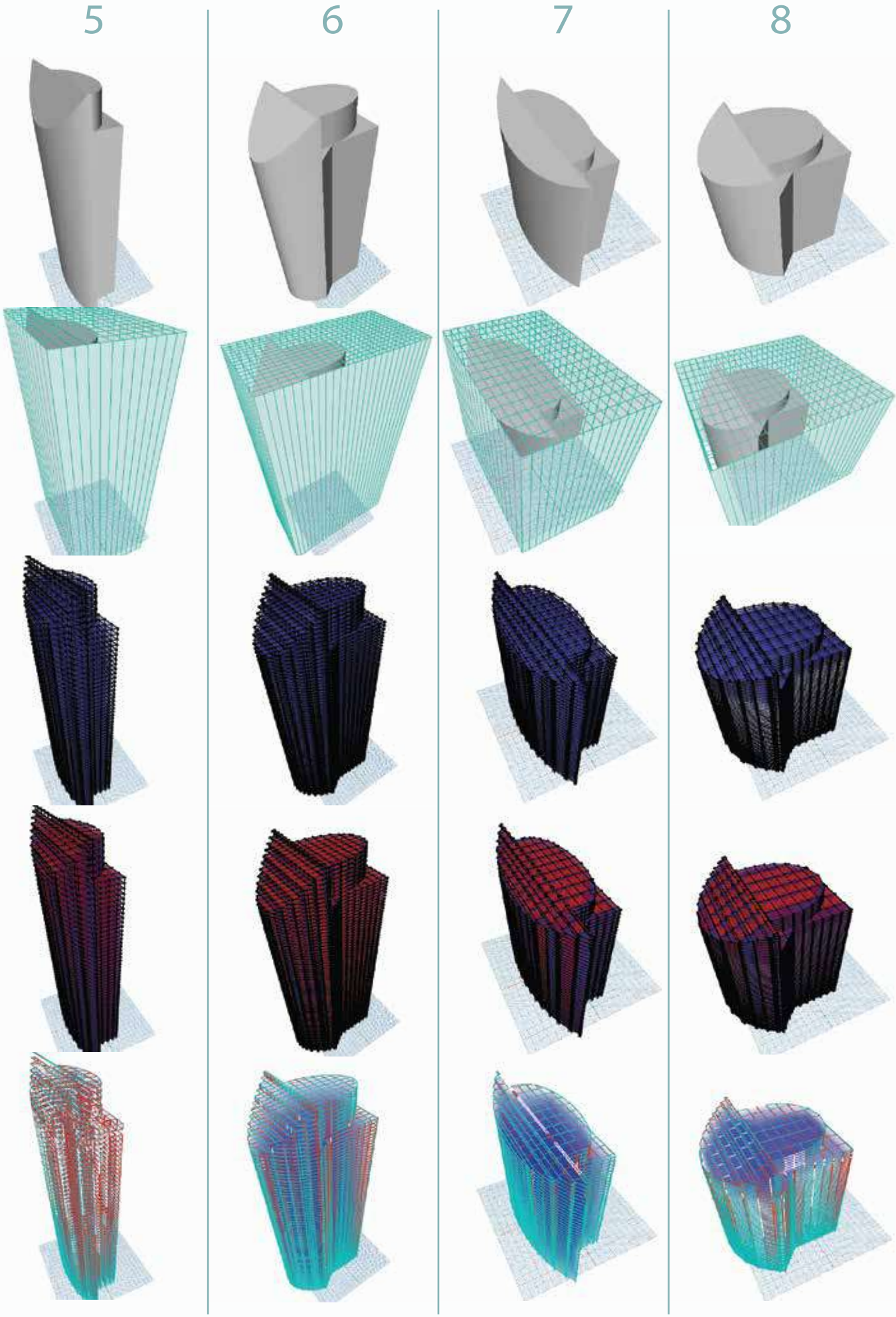


Figura 4.5: Latencia Torre Mayor. Gráfica con escala logarítmica obtenida a partir de los datos de tiempo de ejecución de la tabla de parámetros para la edificación "Torre Mayor". Este es el caso de estudio principal que se tuvo en este trabajo. Algunas cuestiones interesantes son por ejemplo que el modelo #1 se refiere a las dimensiones y estructuración mas cercana a la realidad en base a los bocetos encontrados de la Torre Mayor. En cuanto a la latencia, ocurre un comportamiento similar al modelo rectangular. Se tiene la mayor parte de los modelos alrededor de los diez minutos, sin embargo el modelo de 43 846 Barras se dispara a 45 min, esto es 3 veces más que el modelo anterior que tiene solo la mitad de barras. Además se observa un pico en la gráfica, a pesar de distintas pruebas sigue ocurriendo este comportamiento, y se podría deber a que este modelo en particular necesita de una búsqueda más robusta de elementos exteriores, por lo que retrasa tanto la generación de la estructura como la distribución de losas.

Torre Mayor	Pruebas en Estructuración				Pruebas en Sólido			
Solid Parameters	1	2	3	4	5	6	7	8
X-Dim [m]	18	18	18	18	20	20	35	40
Y-Dim [m]	44	44	44	44	70	70	50	40
Height [m]	225	225	225	225	225	100	100	250
Structure Parameters	1	2	3	4	5	6	7	8
NFloors	59	59	59	59	59	39	59	79
GridSize_X [m]	63.5	63.5	80	50	70	70	126	140
GridSize_Y [m]	70	80	90	70	110	110	110	70
GridSep_X [m]	20	12	6	30	12	15	22	25
GridSep_Y [m]	12	12	6	40	12	15	12	15
Claro en X max [m]	3.18	5.29	13.33	1.67	5.83	4.67	5.73	5.60
Claro en Y max [m]	5.83	6.67	15.00	1.75	9.17	7.33	9.17	4.67
Material Parameters	1	2	3	4	5	6	7	8
Vigas Concreto								
N Section	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8	40: 50X50-4N8
First Floor	1	1	1	1	1	1	1	1
Final Floor	59	59	59	59	59	59	59	59
Columnas Concreto 1								
N Section	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6	59: 70X70-20N6
First Floor	1	1	1	1	1	1	1	1
Final Floor	30	30	30	30	30	30	30	30
Columnas Concreto 2								
N Section	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8	45: 60X60-4N8
First Floor	30	30	30	30	30	30	30	30
Final Floor	59	59	59	59	59	59	59	59
Slab Loads	1	2	3	4	5	6	7	8
live_load [Kg/m2]	100	100	100	100	100	100	100	100
dead_load [Kg/m2]	240	240	240	240	240	240	240	240
Analysis	1	2	3	4	5	6	7	8
Restricted Supports	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Latencia	1	2	3	4	5	6	7	8
Barras	22549	14747	4877	3167	15362	15122	19087	43846
Nodos	8264	5518	1959	1312	5731	5600	7079	15812
Ecuaciones	49584	33108	11754	7872	34386	33600	42474	94872
Tiempo Grid [min]	0.01210	0.00840	0.00708	0.00367	0.00742	0.00778	0.01253	0.02293
Tiempo Structure [min]	2.75038	1.52485	0.41183	0.23890	1.75132	1.94297	2.78623	7.11688
Tiempo Cargas [min]	12.14495	5.03700	0.56270	0.25225	5.67093	8.23868	8.43497	32.75673
Tiempo Análisis [min]	1.41412	0.51357	0.14640	0.08022	0.62080	0.97645	0.98710	5.03622
Total [min]	16.32155	7.08382	1.12802	0.57503	8.05047	11.16588	12.22083	44.93277

Tabla 4.10: Tabla con todos los parámetros necesarios para generar las modelos mostrados utilizando el sólido tridimensional de la "Torre Mayor". Se incluyen los tiempos de ejecución o latencia por cada modelo.


1
2
3
4


5
6
7
8

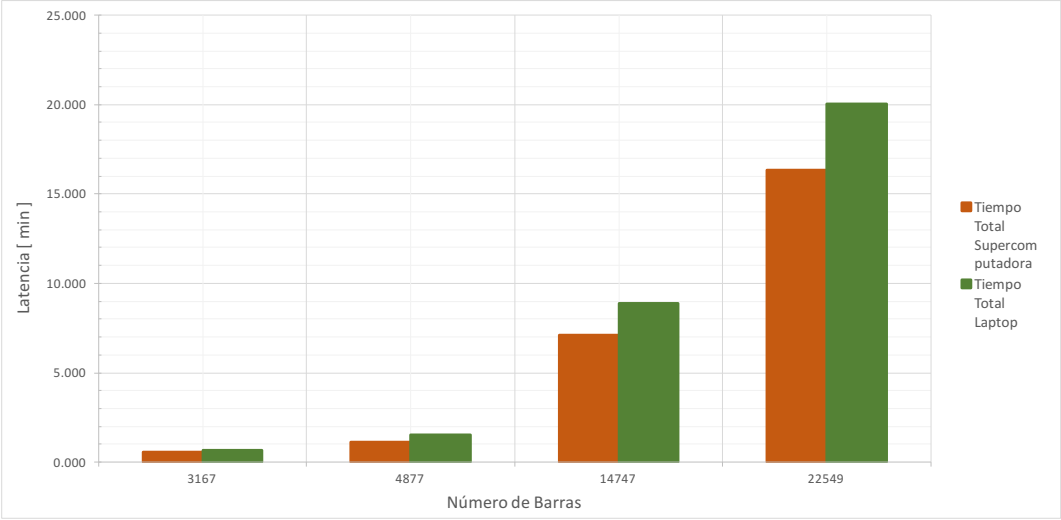


Figura 4.6: Comparación de Latencia Torre Mayor. Gráfica con escala lineal obtenida a partir de la comparación de resultados en el modelo de la"Torre Mayor" al generarse en dos equipos de cómputo distintos. Los modelos que se generaron con el equipo de supercómputo se presentan en color anaranjado mientras que los generados por un equipo de cómputo común tienen color verde

Para finalizar se mostrará una gráfica comparativa de tiempo de actualización o latencia referentes a los primeros cuatro modelos de la "Torre Mayor". Como se comentó antes, los modelos vistos anteriormente fueron generados por una computadora proporcionada por el ***Centro de Supercómputo del Estado de Guanajuato - CIMAT***, pero debido a que los módulos que se programaron en este trabajo están dirigidos a usuarios que poseen computadoras con hardware intermedio, se decidió comparar los tiempos en una computadora tipo laptop. Como podemos observar, no se tiene una diferencia tan considerable de tiempo - para mi sorpresa - ya que se esperaba que fuera una latencia mucho mayor la necesaria por el equipo laptop. Sin embargo aún así, existen alrededor de 4 minutos de ahorro para el modelo de 22549 barras, correspondiente al modelo más realista de la Torre Mayor. Esto tiene un impacto positivo, ya que los códigos implementados no son tan lentos al ejecutarse en un equipo de cómputo alcanzable por cualquier usuario.

4.2. Resultados en GiD

GiD es un sistema CAD de Pre y Post - Proceso desarrollado en el Centro Internacional de Métodos Numéricos en Ingeniería (CIMNE), que incluye una gama amplia de geometrías de alto nivel para definición de un problema físico. Posee un completo conjunto de herramientas para trazado y manipulación de geometría. Además incluye algoritmos de mallado especializados, rápidos y eficientes para realizar análisis numérico avanzado. La ventaja de utilizar este software es que tiene múltiples funciones de visualización de resultados numéricos de problemas físicos (17). Debido a las ventajas que este programa tiene para la visualización de resultados y a que el software de análisis estructural **MECA** tiene la posibilidad de exportar resultados en el formato de **GiD**, se mostrarán a continuación una serie de imágenes que presentan el comportamiento de algunas edificaciones dentro del post-procesador de este software.

Como ya se ha hablado durante este trabajo, el uso de la parametrización beneficia en el hecho que se pueden obtener múltiples edificaciones con tan solo algunos cambios en parámetros. Esto es un aspecto importante, ya que se puede dar diversas soluciones estructurales a un diseño arquitectónico dado. Además, es posible generar el modelo indicado y eficiente de acuerdo a valores de resistencia mecánica o incluso a costos de material. El siguiente conjunto de imágenes muestra un caso particular de una edicificación con forma poligonal de cinco lados, es decir un pentágono. Dicho modelo fue probado por distintas configuraciones estructurales con el motivo de encontrar un modelo que tuviera menor efecto mecánico, por ello se utilizó **GiD** como post-proceso para observar los desplazamientos, momentos internos y eficiencias de cada elemento estructural. Cabe resaltar que los resultados numéricos no fueron parte de una medición exacta, sino que se generaron los modelos de manera ilustrativa para poder mostrar lo sencillo que se puede realizar una optimización utilizando el modelado paramétrico.

Para finalizar, se mostrará la visualización de resultados correspondiente al modelo más realista realizado para la *Torre Mayor*. Se utilizará el post-proceso que tiene incorporado **GiD** con todos los resultados que pueden mostrarse en este software con relación al análisis estructural matricial utilizando el programa **MECA**. Entre éstos se incluye la verificación de la asignación de materiales que se realiza utilizando los módulos programados, y que podemos observar que se definieron correctamente, también se observa la asignación de cargas a los elementos estructurales y su valor numérico que es producto de la distribución de fuerzas, además se observan los momentos internos de las barras, así como las eficiencias y finalmente se presente el modelo de la Torre Mayor con su estado deformado y antes de deformar, indicando con una escala de color el valor máximo, que para este caso es de aproximadamente 3 cm y localizado en su extremo superior.

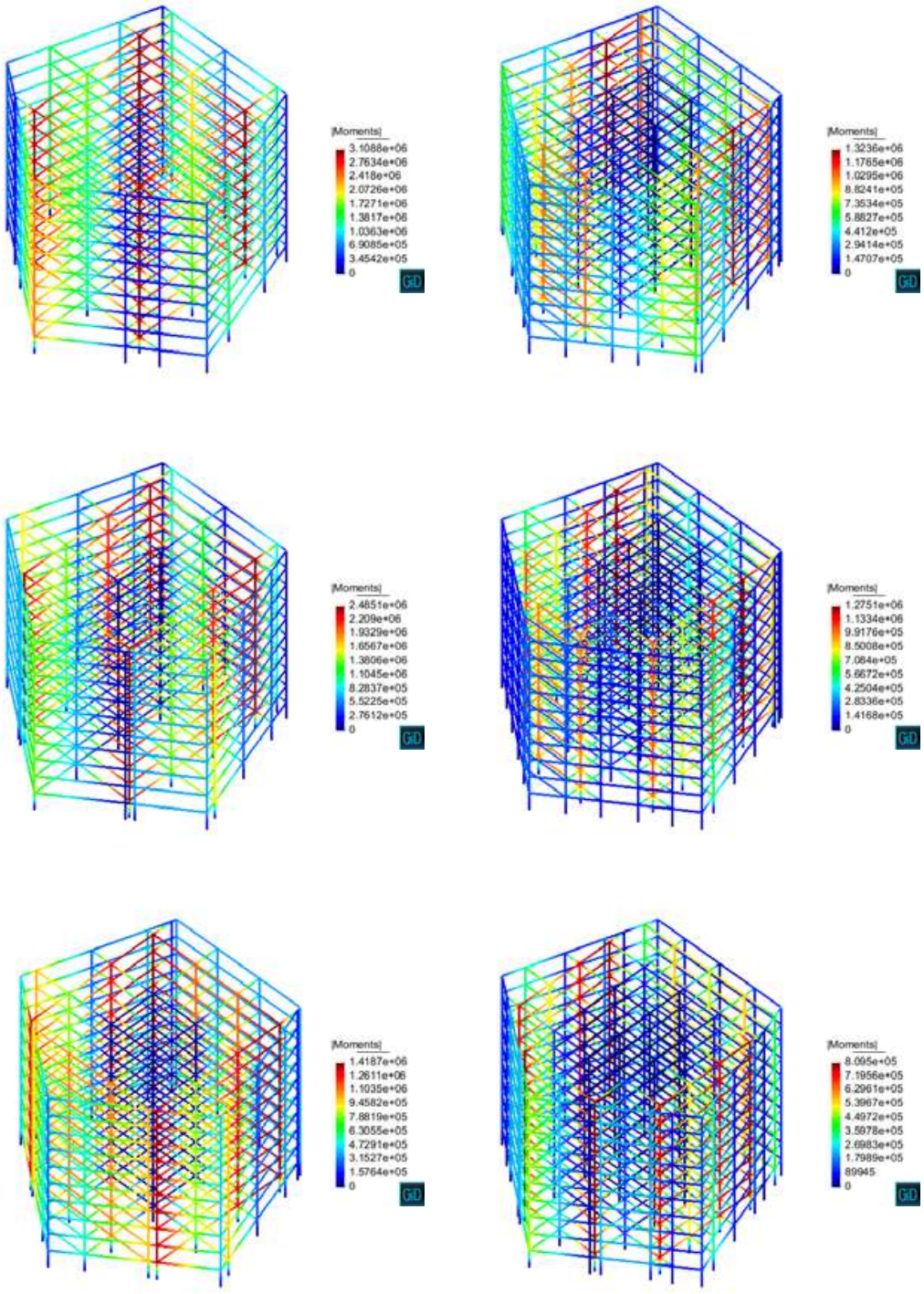


Figura 4.7: Momentos Internos - GiD. Se observa una serie de resultados del edificio con base pentagonal, la cual tiene en una escala de colores con los valores de momentos internos en cada barra, se puede apreciar una reducción del valor numérico de éstos al incorporar más barras en el modelo estructural.

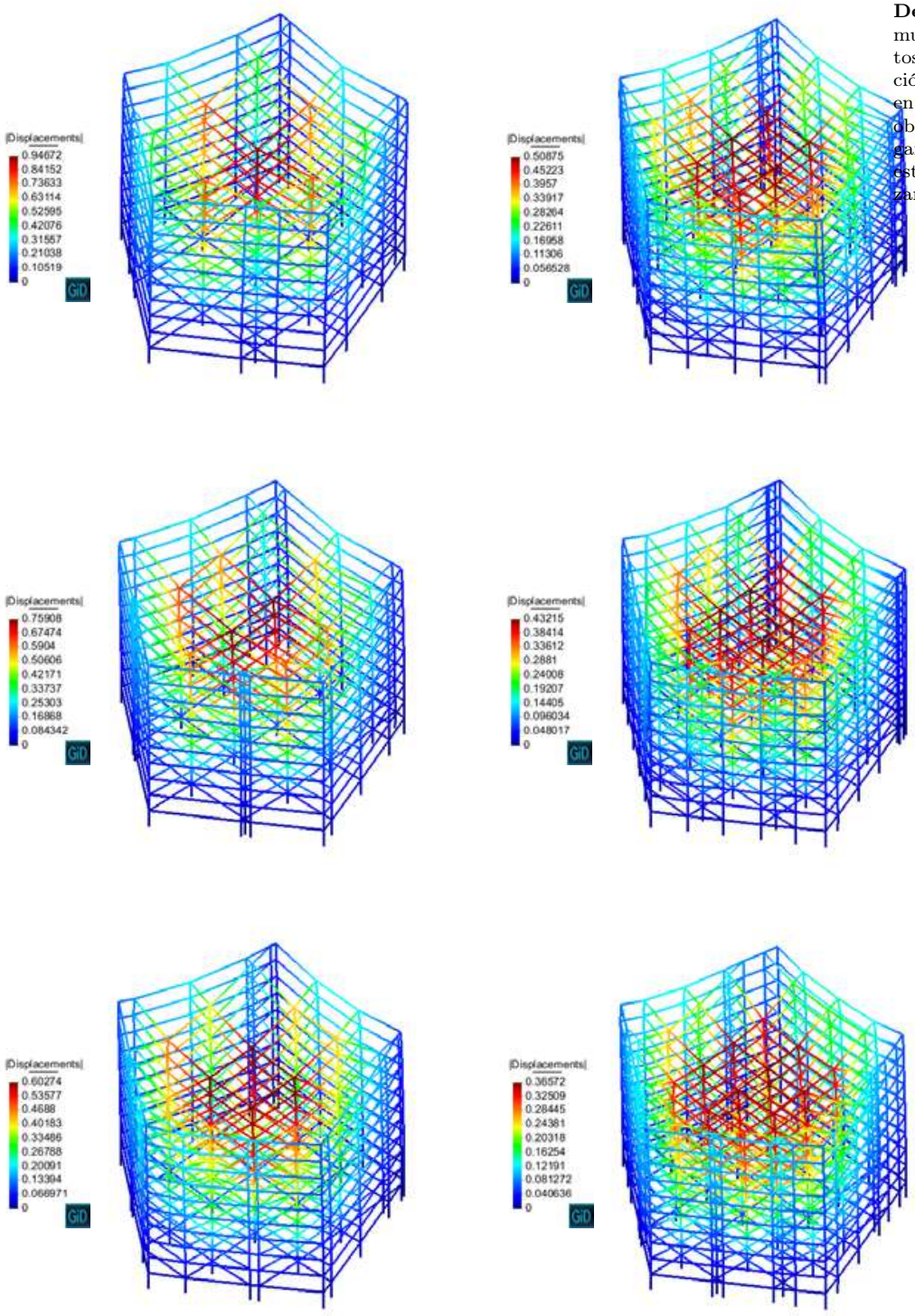


Figura 4.8: Reducción de Desplazamientos - GiD. Se muestran seis modelos distintos para una misma edificación, los cuales se encuentran en estado deformado. Podemos observar que mientras se agregan barras en la configuración estructural, el valor de desplazamiento disminuye.

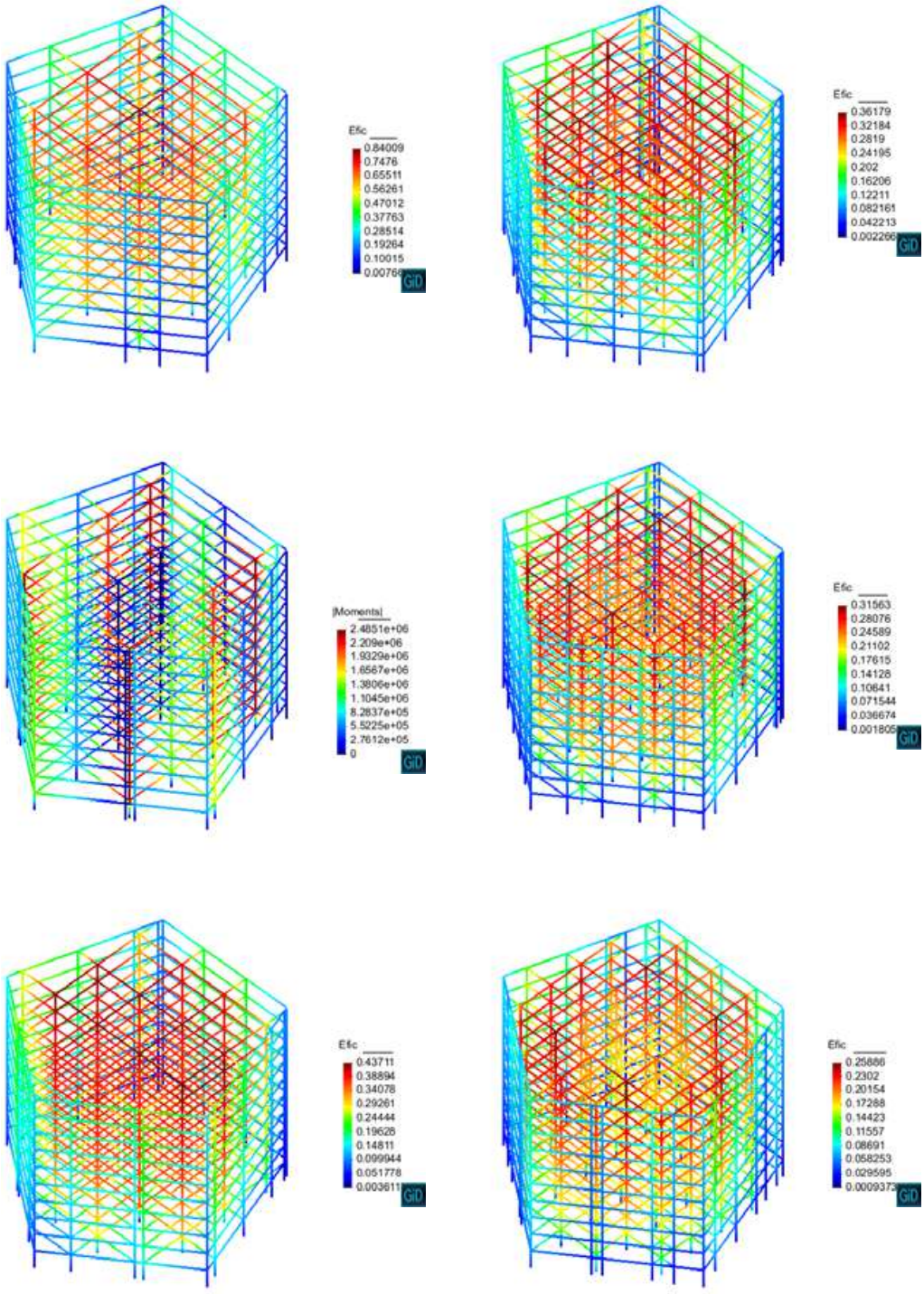


Figura 4.9: Reducción de Eficiencias - GiD. Se muestra los seis modelos de edificación con forma pentagonal, en donde el valor de eficiencia mecánica se reduce debido al incremento de barras.

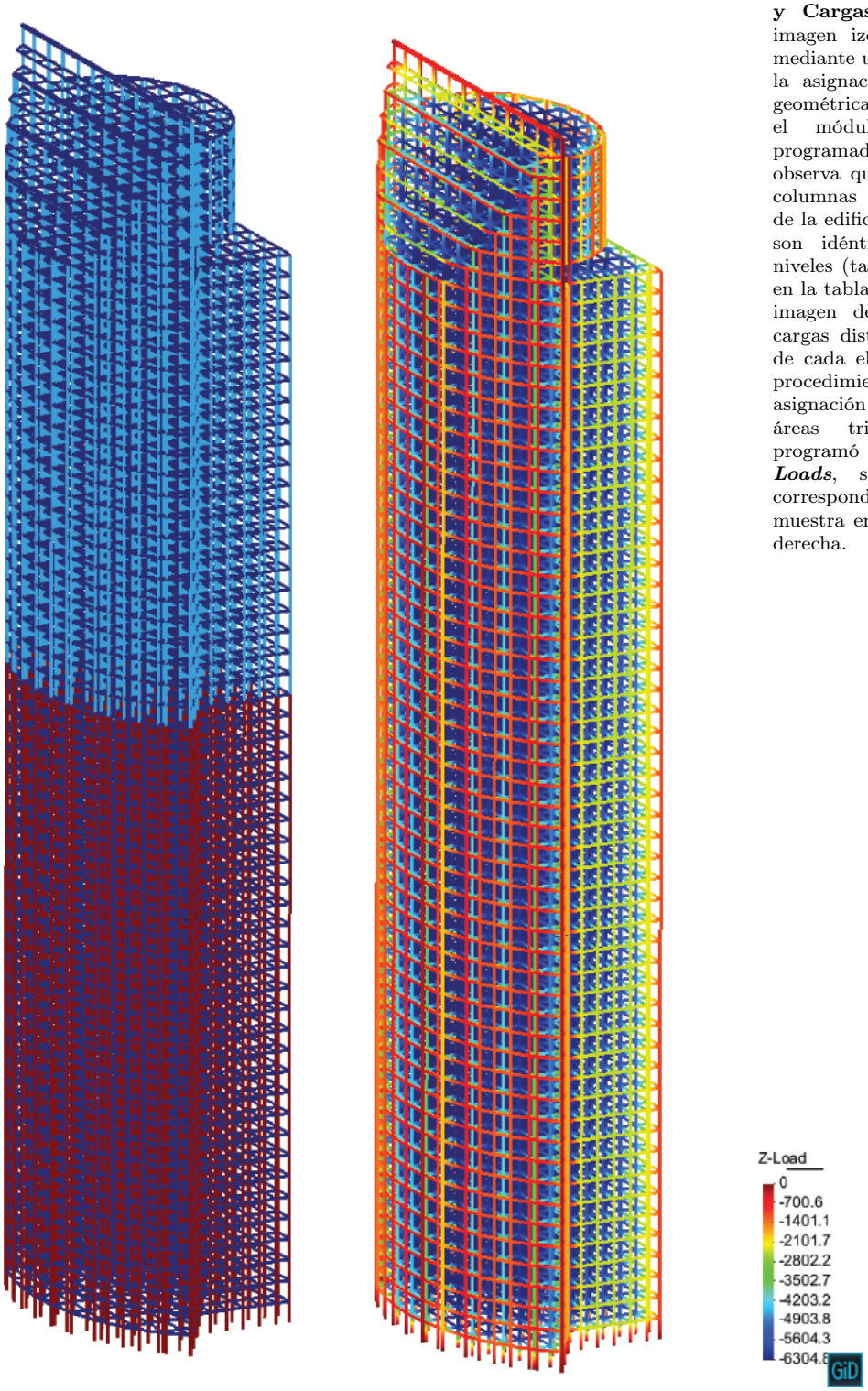


Figura 4.10: Asignación de Secciones/Materiales y Cargas - GiD. En la imagen izquierda se muestra mediante una escala de colores la asignación de propiedades geométricas realizada desde el módulo de materiales programada en este trabajo, se observa que los materiales en columnas difieren a la mitad de la edificación pero las vigas son idénticas en todos los niveles (tal como se menciona en la tabla de parámetros). La imagen derecha muestra las cargas distribuidas a lo largo de cada elemento estructural, procedimiento posterior a la asignación de cargas mediante áreas tributarias que se programó en el módulo ***Slab Loads***, su valor numérico correspondiente al color se muestra en la leyenda inferior derecha.

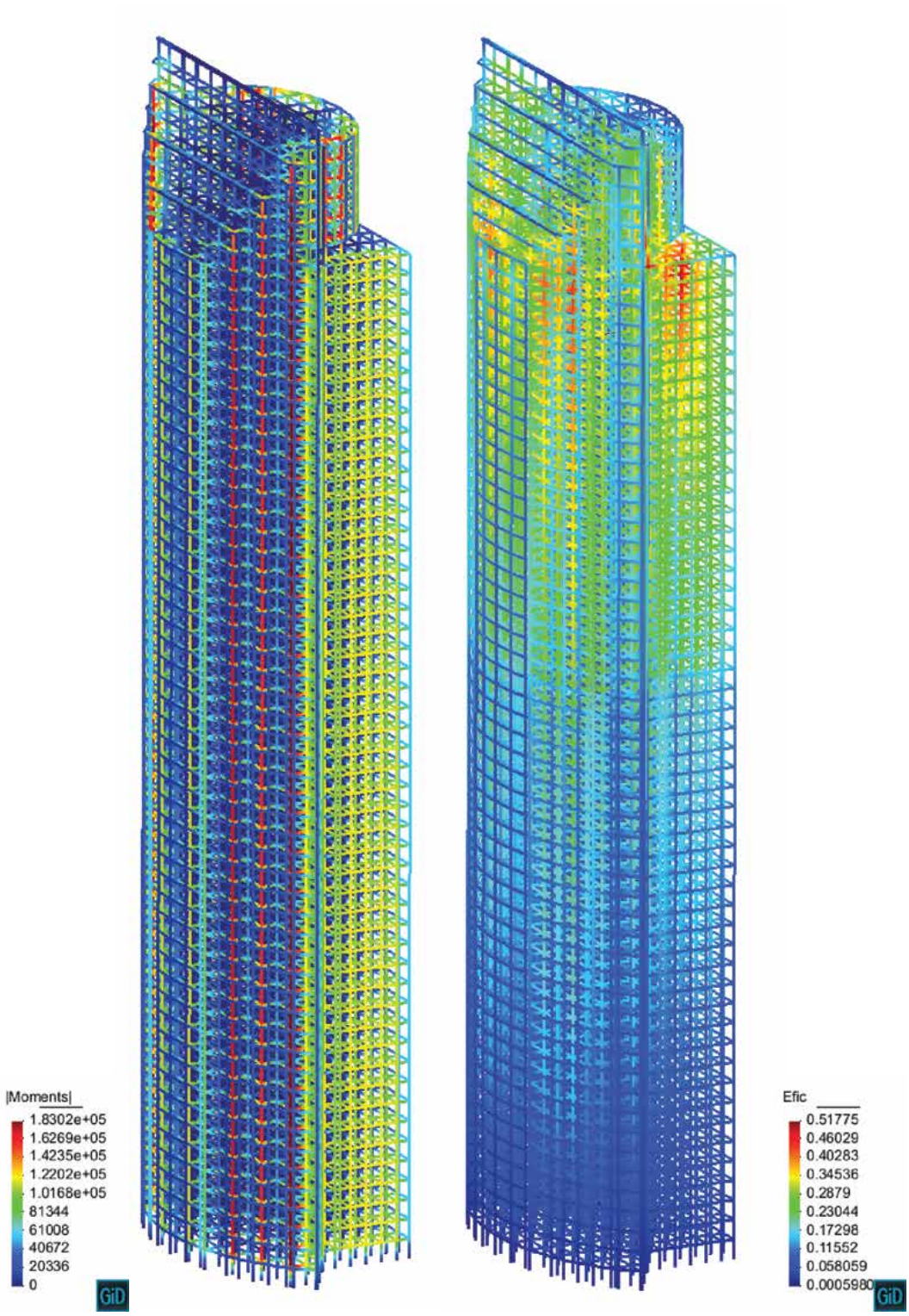


Figura 4.11: Análisis de Momentos y Eficiencias - GiD. La imagen izquierda muestra los momentos en tres dimensiones, se puede observar que se presenta un comportamiento simétrico, por lo que los resultados son congruentes a lo que se modelo utilizando los módulos programados. En la imagen derecha se observan las eficiencias, este resultado está incorporado en los módulos de visualización dentro de **Dynamo**, sin embargo se colocó aquí para mostrar que con GiD se obtienen los mismos resultados.

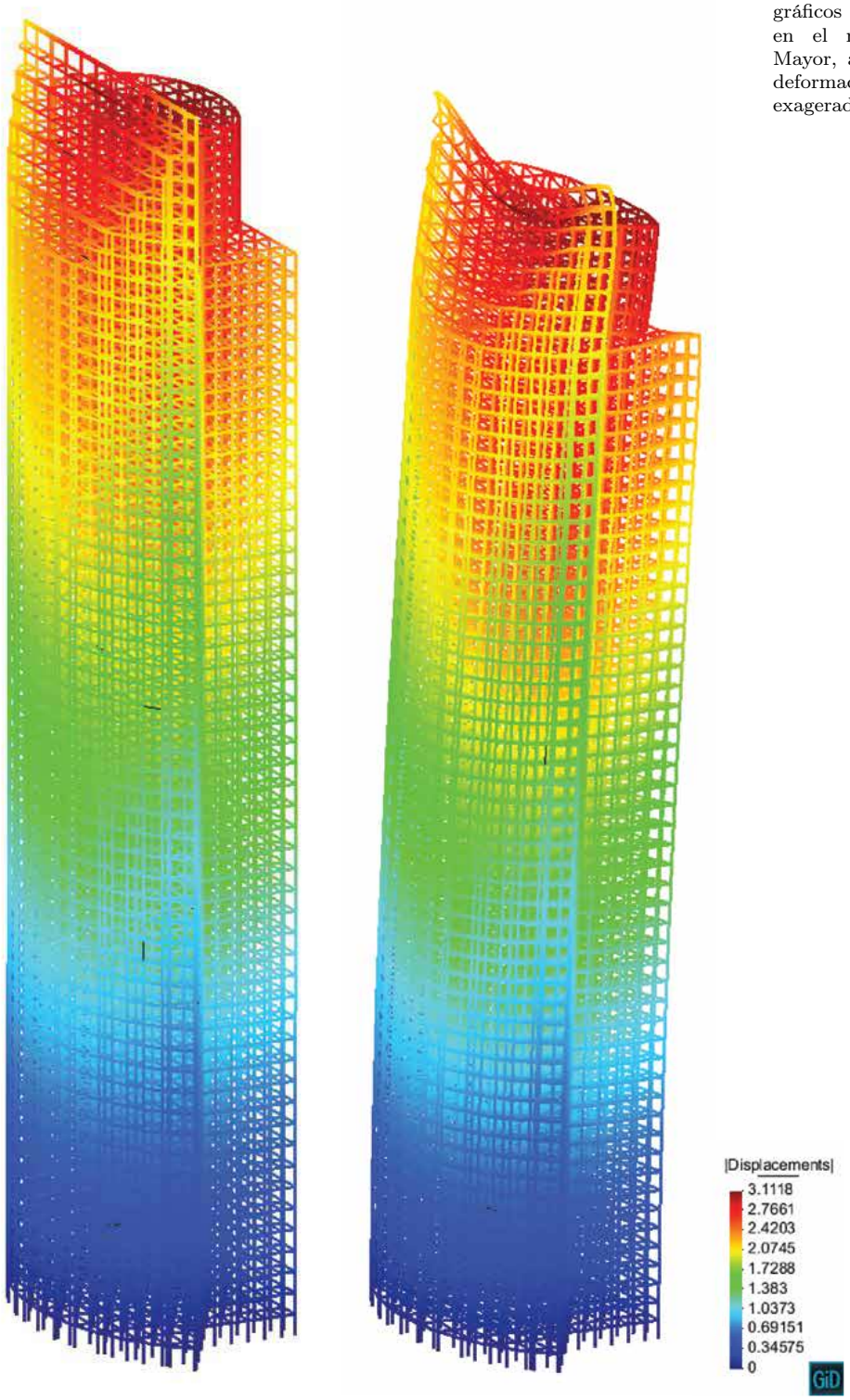


Figura 4.12: Desplazamientos - GiD. Resultados gráficos de desplazamientos en el modelo de la Torre Mayor, además se muestra la deformación con una escala exagerada para su observación.

4.3. Modelos con Alto Número de Barras

Debido a que se está utilizando un equipo de supercómputo solicitado por ***Centro de Supercómputo del Estado de Guanajuato - CIMAT***, se decidió probar algunos modelos que aunque son imposibles de construir en la vida real, computacionalmente se pueden generar y analizar utilizando los módulo programados y presentados en este trabajo. La visualización del post-proceso se hizo mediante el software **GiD** debido a que **Dynamo** no puede mostrar tal cantidad de información e incluso falla al intentar hacerlo. Los resultados mostrados corresponden a las cargas asignadas, desplazamientos, deformación y momentos internos de la estructura.

Los modelos probados en el "cluster" se acomodaron en las páginas siguiendo un aumento de barras, como se observará a continuación, los modelos que son más grandes comenzarán a verse como un sólido debido a que la densidad de barras es demasiada.

Tal como se comentó anteriormente, los tiempos de ejecución presentan un incremento súbito a partir de las $35,000$ barras aproximadamente, por ello los modelos robustos que se probaron en la supercomputadora tienen una actualización aproximadamente de un día o incluso más, será necesario hacer una investigación más detallada en el proceso de almacenamiento de entidades geométricas altas como lo son las superficies y los sólidos, además del recurso computacional utilizado por el paradigma de programación visual; esto ayudaría a conocer cuál es el motivo del tiempo requerido para generar la estructuración y que solución se podría dar, sin embargo este problema corresponde al área de Ciencias de la Computación, por lo que se dejará para un trabajo futuro.

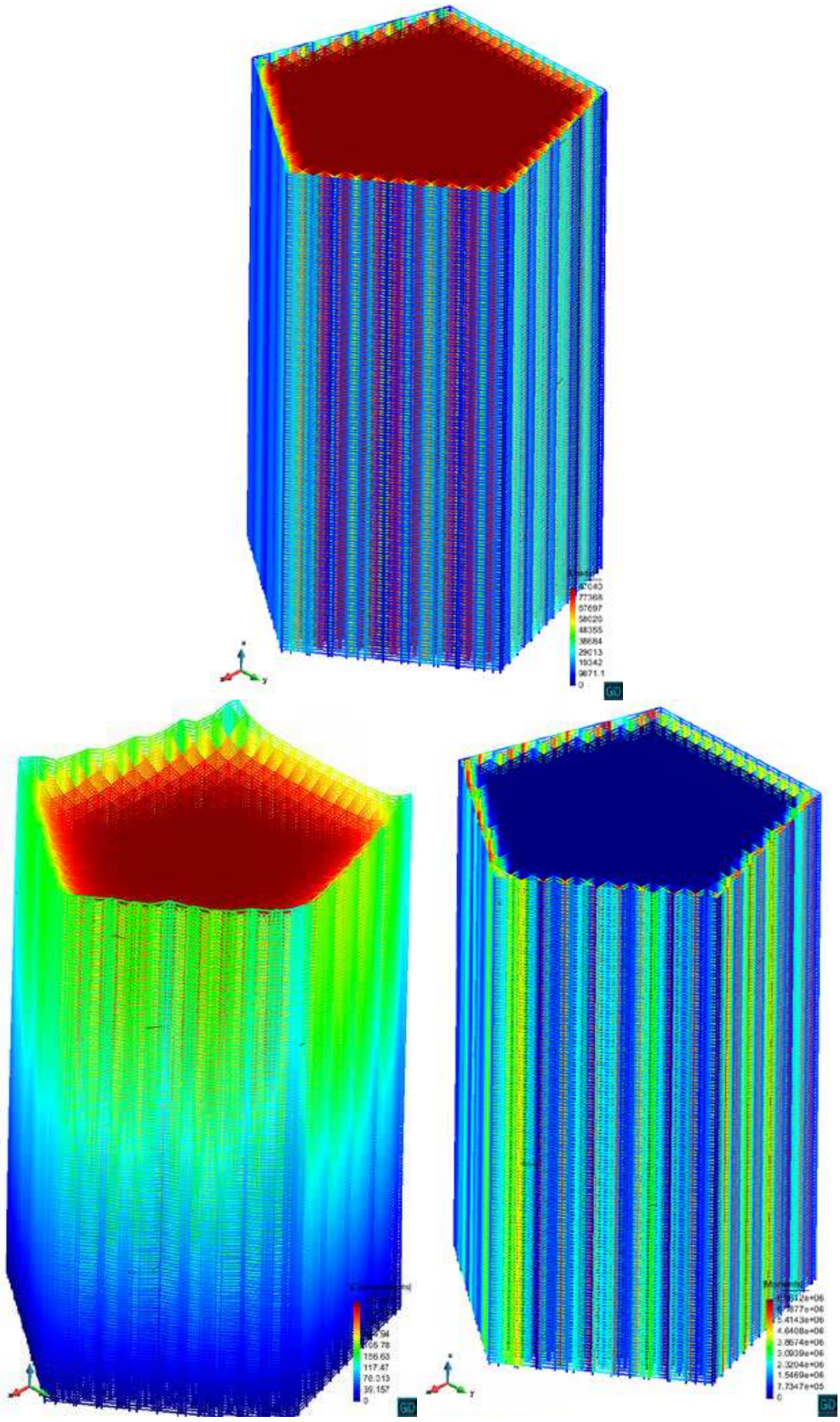


Figura 4.13: Edificación Pentagonal. Se muestran los resultados de asignación de cargas, momentos internos y desplazamientos de una edificación con 202,650 Barras y 70,517 nodos, lo que representa resolver un sistema de 423,102 ecuaciones numéricamente.

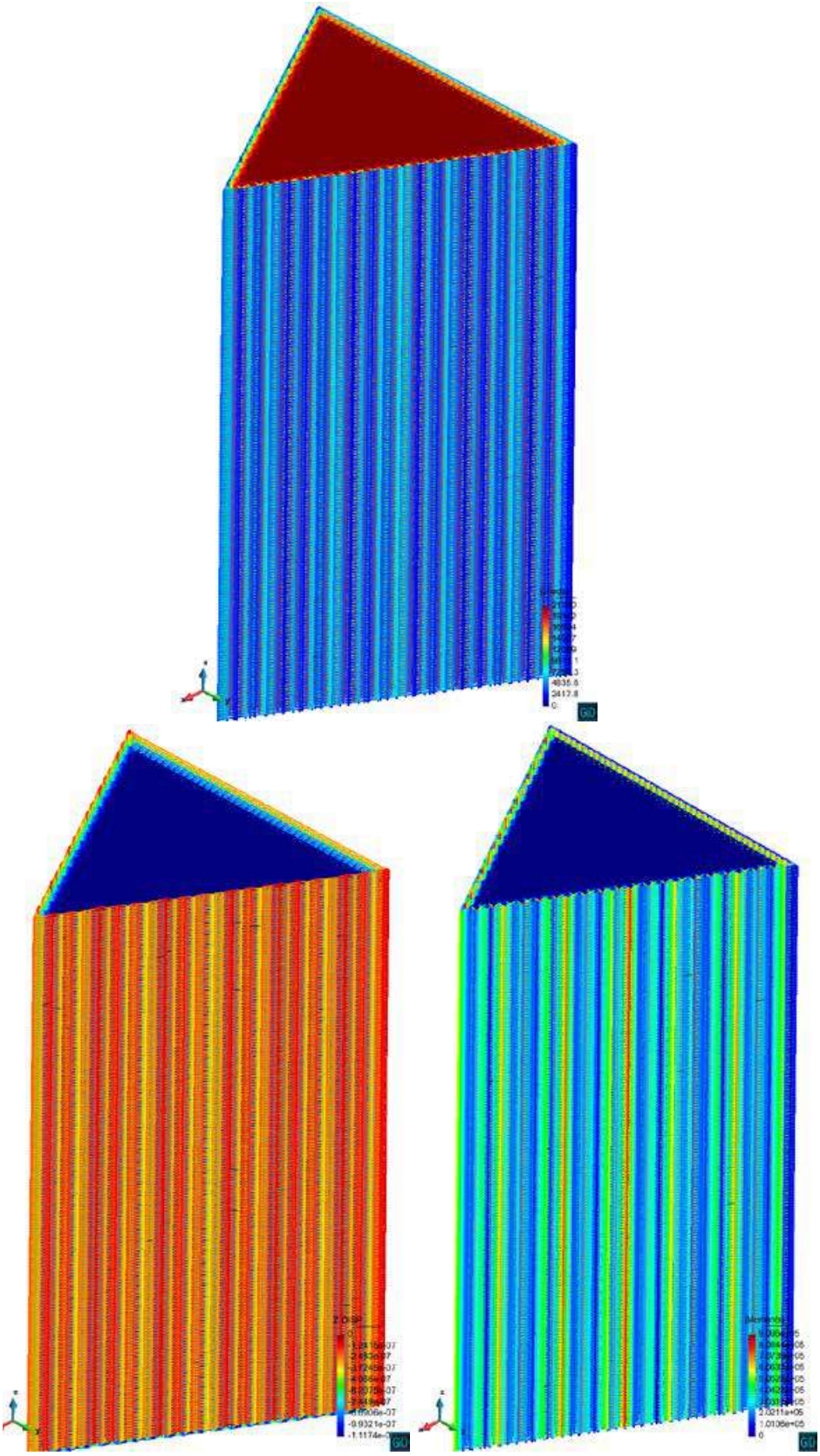


Figura 4.14: Edificación Triangular. Se muestran los resultados de asignación de cargas, momentos internos y desplazamientos de una edificación con 568, 600 Barras y 195, 975 nodos, lo que representa resolver un sistema de 1,175,850 ecuaciones numéricamente. En este caso los desplazamientos son prácticamente cero, debido a esto la estructura deformada no parece, siendo que se tiene un factor de escala de 1, 000, 000, 000.

4.4. Discusión

Como resumen de los resultados, podemos decir que el principal producto de este trabajo fue el desarrollo de una herramienta que pretende reducir el costo de hacer cambios durante la fase de planeación para una propuesta estructural. La metodología empleada, denominada *Diseño Paramétrico*, tiene ventajas sobre un procedimiento tradicional debido al tiempo que toma construir un modelo estructural. Sin embargo, debido a que esta metodología tiene una expansión reciente, aún no se tiene conocimiento concreto de como medir su eficiencia. La flexibilidad que se obtiene al programar visualmente una aplicación como esta es un punto fundamental y por ello se tomó como referencia las mediciones propuestas por Davis (9), y que son base en la estandarización para el desarrollo de software. Pero debido a la particularidad de los programas paramétricos, las características mencionadas al principio de este capítulo considero son de caracter subjetivo aún. Por ello no es posible tener una respuesta precisa si una aplicación es mejor que otra, lo único que se puede hacer es comparar la utilidad de la aplicación frente a un problema propuesto. Quizá la medición más importante en este trabajo fue el tiempo de actualización o latencia, ya que éste le da al algoritmo una ventaja frente a otros procedimientos. Los resultados mostrados en las gráficas de latencia con respecto al número de barras es una muy buena forma de conocer como se comporta este paradigma de programación que se basa en la ejecución del código no de forma estructurada, sino que el procesador dirige que parte se ejecuta. Y lo que podemos observar es que para los modelos con menos barras, el rendimiento es muy bueno ya que en menos de uno o dos minutos, se tiene completamente generada la estructura de un edificio, así como su análisis. Pero este comportamiento presenta un límite, ya que para más de 35 000 barras aproximadamente, el tiempo necesario crece súbitamente, esto se piensa que es por el recurso computacional utilizado, ya que el número de componentes geométricos tridimensionales crece y éstos requieren de mucha memoria para su almacenamiento. La otra medición certera es simplemente que la aplicación haga su trabajo, es decir que tanto la generación de la estructura, la distribución de cargas, el análisis estructural y la verificación de resistencias se ejecuten de manera adecuada. Esto lo comprobamos con las imágenes mostradas, y como podemos observar es muy sencillo proponer múltiples soluciones para una edificación, esto da pie a realizar una **optimización**, tan solo en esta sección se muestran 8 propuestas por cada edificación. Sin embargo, aún falta mucho por hacer, se recomienda continuar con este trabajo para incrementar la flexibilidad de la aplicación. Lo que significa agregar distintos elementos estructurales en base a procesos constructivos y módulos de diferentes tipos de cargas.

Futuro en Ingeniería Estructural. Edificación en construcción denominada World Trade Center 3 de la firma Rogers Stirk Harbour + Partners, dentro de la zona económica de Nueva York. (E. Muttio - 2016)

Capítulo 5

Conclusiones

Los proyectos de construcción se están volviendo cada vez más complejos. Se requiere mucho trabajo de logística y manejo de recursos para poder entregar un producto confiable y en el tiempo que se solicita, lo cual es común que en proyectos de construcción los periodos de entrega son muy cortos. El desarrollo de herramientas a través del tiempo ha beneficiado a los proyectistas, como lo menciona Sacks (41), ya han existido revoluciones en cuanto a los procesos. La primer revolución fue el uso del sistema CAD en la industria de la arquitectura, ingeniería y construcción o por sus siglas en inglés AEC, los diseñadores se movieron de un dibujo a mano al dibujo asistido por computadora. Esta revolución está completa, y como ésta práctica fue adoptada, nuevas versiones de software cada vez reducen más el tiempo de planeación. Ahora la segunda revolución CAD es el modelado de sólidos paramétricos, ya que los beneficios potenciales son dos. De manera directa (reducir costos de diseño y dibujo, habilitando la producción automática) y de manera indirecta (reducir las tasas de error en la construcción, brindando la habilidad de considerar alternativas de diseño).

5.1. Diseño Paramétrico vs Proceso Tradicional

El resultado principal de este trabajo es la programación de un conjunto de módulos que están dentro de un contexto de programación visual, los cuales se encuentran ya incluidos dentro de una librería de un software paramétrico comercial y pudieran ser distribuidos para ser utilizados como herramienta para Ingeniería Estructural. El problema que se atiende usando este conjunto de módulos es la estructuración de una edificación de manera sencilla, que puede tener distintas formas geométricas, y que puede ser analizado y evaluado por el mismo programa. Esto da muchos beneficios, ya que los tiempos de entrega son limitados, los ingenieros no disponen de tiempo para que los proyectos tengan un grado de optimización aceptable o carecen de diversas soluciones a un

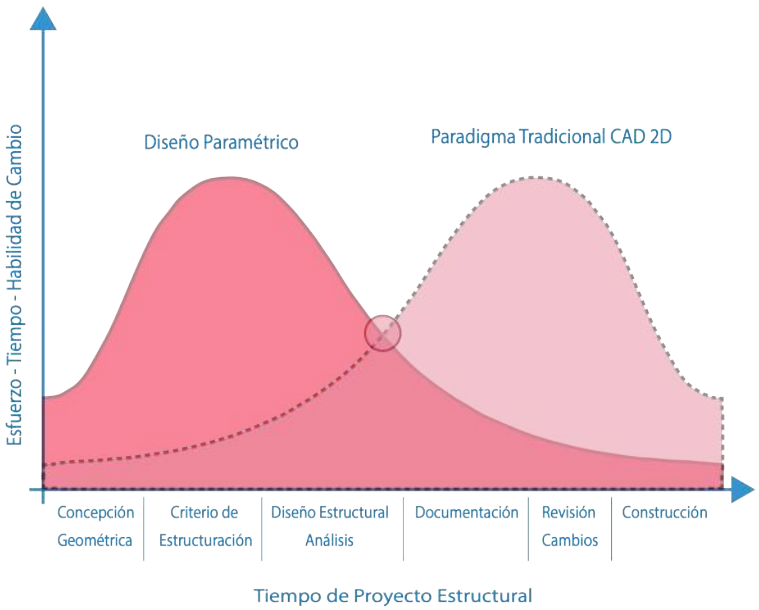


Figura 5.1: Diseño Paramétrico vs Proceso Tradicional. Gráfica empírica basada en las curvas de Paulson y Macleamy, presentadas por Davis ((9)), en la cual se observa en el eje vertical la cantidad de esfuerzo y tiempo invertido en diferentes fases de un proceso estructural.

problema. Con la práctica tradicional se busca la manera de que el diseño propuesto funcione, y por tal motivo algunos proyectos son sobre-diseñados o con otras palabras, su resistencia es por mucho más alta que las solicitaciones. El diseño estructural paramétrico puede ser una de muchas soluciones para la reducción de esfuerzo y tiempo. Además el diseñador estructural puede proponer muchas soluciones a una edificación y hacer una selección de acuerdo a la más adecuada para el proyecto, lo cual ya se puede decir que se está aplicando un proceso de optimización aunque sea de manera simple.

De acuerdo a lo mostrado en los capítulos 3 y 4 acerca del funcionamiento y resultados obtenidos del modulo de estructuración, podemos observar que la alternativa del uso de parametrización de procesos, incluyendo en esta una automatización de los mismos, tiene un efecto positivo en cuanto a los pasos donde se requiere realizar constantes modificaciones. El uso de herramientas de dibujo como lo son los sistemas CAD, son una buena herramienta cuando se desea proyectar, sin embargo es bien sabido que el tiempo empleado para el dibujo es alto, incluso se debe invertir en personas que dediquen todo su tiempo en este proceso. La figura 5.1 muestra una gráfica empírica adaptada del uso de modelos paramétricos con la inclusión de una correcta gestión de información, lo que es llamado Modelos de Informacion en la Construcción o por sus siglas en inglés “BIM” (9). Esta gráfica muestra las posibles áreas de aprovechamiento utilizando las herramientas computacionales como lo son el diseño paramétrico y los sistemas CAD. Como podemos observar en la curva, el siguiente paso a encontrar es la relación entre el tiempo, esfuerzo y habili-

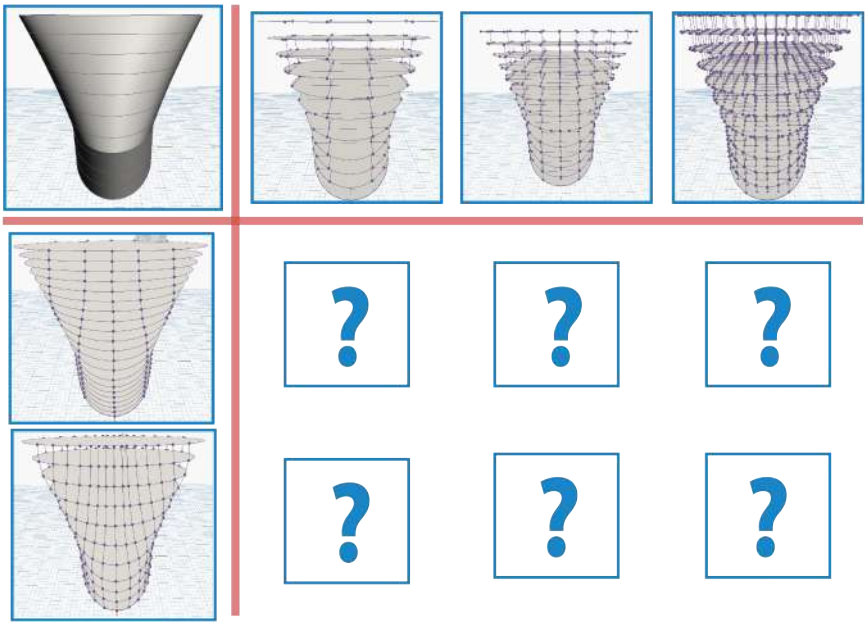


Figura 5.2: Optimización Estructural. Matriz de solución para una edificación que podría ser resuelta por un algoritmo de optimización para una solicitación en particular.

dad de cambio utilizando los dos paradigmas de diseño. Aunque la curva no tiene resultados cuantitativos, podemos inferir diversos comportamientos con lo aprendido utilizando el módulo de estructuración.

5.2. Trabajo Futuro

Cuando se desea realizar proyectos que tienen ciertos puntos repetitivos, es cuando podemos buscar alternativas para hacer más eficiente el trabajo. Otro aspecto importante a considerar, es que las propuestas mediante un proceso tradicional no siempre son tan óptimas como podrían serlo. Es decir, no solo se busca corregir errores de diseño cuando éste no cumple la reglamentación mínima, sino que muchos proyectos pueden reducir sus costos al optimizarlos. Cuando se realiza un proyecto de construcción se requiere mucho trabajo para poder presentarlo en la fecha solicitada, debido a que es comun que sean cortos los periodos en los que se debe entregar la documentacion necesaria. Por lo que un proceso de optimización ligado a la parametrización es fundamental, por ello que el siguiente nivel de este trabajo es el desarrollo de una técnica computacional que ayude en cuanto a la selección del mejor candidato para un diseño en específico. De esta manera se podrían encontrar estructuraciones que permitan reducir el volumen de los materiales empleados, como por ejemplo el concreto y el acero, y por lo tanto se reducirá el costo total de la obra.

Herramientas como ésta ofrecen alternativas que mejoran el desempeño de un proyectista a la hora de proponer soluciones estructurales y que además permite la modificación de parámetros en corto tiempo para adecuarse a los cambios repentinos del mismo. Los resultados presentados en este trabajo nos permiten imaginar en la gama de posibilidades de una sola edificación, por lo que el ingeniero ahora sí podría pensar en optimizar el diseño. Ya que teniendo "en el momento" los resultados del análisis estructural y en base a la generación de muchas propuestas se puede tener la seguridad de satisfacer la necesidad del proyecto ofreciendo diseños óptimos e innovadores.

Apéndice A

Implementación en Python

En este apartado se coloca el código textual implementado en los diferentes módulos que se programaron para la estructuración, distribución de cargas gravitacionales y análisis estructural dentro de la paquetería nombrada "Structural Dynamo". Se utilizó un lenguaje de alto nivel denominado *Python* y la librería de geometrías que proporciona el software de modelado paramétrico **Dynamo**. La forma en como esta organizado este apéndice es siguiendo el flujo de las funciones como se detalla en el capítulo cha:metod, por lo que se recomienda leerlo en conjunto con este apéndice.

A.1. Código: Torre Mayor

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math

#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN

def get_chaplane(rad_x,rad_y,height,offset):

  #Solo funciona para el plano YZ, se debe encontrar la forma de automatizar con
  #condicionales para cualquier otro plano

  #Falta anexar la formula de la elipse para encontrar la distancia real del elipse

  #Obtencion de puntos que generan el plano
  #Se implemento la funcion que genera un semielipse mediante geometria analitica para
  #conseguir la distancia real en eje Y
  #de los puntos, ya que al cortar el semielipse, las longitudes no son las especificadas

  real_y=math.sqrt((1-(offset*offset)/(rad_x*rad_x))*(rad_y*rad_y))

  p1=Point.ByCoordinates(offset*0.05,real_y,height);
  p2=Point.ByCoordinates(offset*0.05,-real_y,height);

  if offset<0:
    p3=Point.ByCoordinates(-rad_x-offset,0,0);

  else:
    p3=Point.ByCoordinates(rad_x-offset,0,0);


  points=[]
  points.append(p1)
  points.append(p2)
  points.append(p3)


  #Obtencion de vectores para generar el plano y su normal que define la direccion
  #para la utilizacion de la funcion trim que necesita conocer la parte del solido a descartar

  v1=Vector.ByTwoPoints(p1,p2)
  v2=Vector.ByTwoPoints(p1,p3)

  if offset>=0:
    v_normal=v1.Cross(v2)

  else:
```

```
    v_normal=v2.Cross(v1)

  #Obtenci n del plano de corte para el chafl n
  cha_plane=Plane.ByOriginNormal(p3, v_normal)

  return cha_plane,p3

#Se restringe el plano de corte a los planos YZ y XZ que proporciona Dynamo

def sem_elli_YZ_cha(rad_x,rad_y,height, offset):
  """
  Descripci n:
    Genera un s lido semiel ptico con chafl n mediante la generaci n de un elipse
    haciendo un corte con ayuda de un plano YZ, adem s se puede modificar la posici n
    en X del plano para obtener un s lido de distinta geometr a. La posici n del
    s lido se encuentra en el centroide del primer elipse como origen, por lo que el
    semielipse se encuentra partiendo del lado recto en el origen coordenado, se
    incluyo una funci n que permite trasladar al origen dicho s lido al cambiar la
    posici n del plano de corte. Es importante mencionar que el double "offset" debe
    estar contenido en el rango -rad_x < offset < rad_x para que funcione de manera
    adecuada, dependiendo el signo que tome esta variable, el elipse cambiar de
    cuadrante. El chafl n que se genera se realiza mediante un plano que corta
    uniendo los puntos en extremo superior del semielipse y un punto central
    inferior.
    Los puntos son generados mediante la funci n get_plane() para posteriormente
    generar el plano de corte del chafl n.

  Argumentos:
    rad_x: radio en direcci n de la coordenada X.
    rad_y: radio en direcci n de la coordenada Y.
    height: n mero que funge como propiedad del solido y que determinar
      la separaci n entre planos.
    offset: n mero que define la posici n del plano de corte en el eje X,
      con rango -rad_x < offset < rad_x

  Variables:
    Identity: Sistema coordenado en el origen.
    xc: Variable usada para definir la coordenada X para el sistema coordenado en
      el punto m s alto.
    yc: Variable usada para definir la coordenada Y para el sistema coordenado en
      el punto m s alto.
    tall_pt: tallEST_pOINt es un punto utilizado para definir el sistema coordenado
      m s alto.
    cs: cOORDINATEsYSTEM es el sistema coordenado en el punto m s alto del s lido.
    elli_1: elliPSE_1 curva inferior en forma de elipse utilizada para generar el
      s lido

    elli_2: elliPSE_2 curva superior en forma de elipse utilizada para generar el
      s lido

    elli_vec: elliPSE_vecTOR lista que contiene las dos curvas de elipses para
      generar el s lido.
    elli_sol: elliPSE_solID S lido con forma de elipse completo.
    cut_plane: Plano de corte de elipse.
    aux_vector: vector auxiliar normal para generar el cortado del s lido.
    aux_point: Punto necesario para realizar la operaci n del trim, se obtiene este
      punto para conocer que secci n es la que se necesita.

  """

  #Creaci n de Sistema Coordenado en el origen

  Identity=CoordinateSystem.Identity();

  #Creaci n de Sistema Coordenado en punto m s alto

  xc=0;
  yc=0;

  tall_pt=Point.ByCoordinates(xc,yc, height)


  cs=CoordinateSystem.ByOriginVectors( tall_pt, Identity.XAxis, Identity.YAxis, Identity.ZAxis)

  #Creaci n de elipses

  elli_1=Ellipse.ByCoordinateSystemRadii(Identity, rad_x, rad_y)

  elli_2=Ellipse.ByCoordinateSystemRadii(cs, rad_x, rad_y)

  elli_vec=[]
  elli_vec.append(elli_1)
  elli_vec.append(elli_2)

  #Creaci n de s lido completo

  elli_sol=Solid.ByLoft(elli_vec)

  #Traslado de la geometr a al origen (desde python script no puedo trasladar el trim, tengo que trasladar la figura antes de cortarla)

  elli_sol=elli_sol.Translate(-offset,0,0)


  #Trim del s lido
```

```
  #Plano de corte, se determina mediante el plano (YZ) y con un offset que puede ser desde - rad_x hasta + rad_x

  cut_plane=Plane.ByOriginNormal(Point.ByCoordinates(0-offset,0,0),Vector.ByCoordinates(1,0,0));


  cut_plane=cut_plane.Offset(offset);

  #Punto Auxiliar
  #Se busca mediante el vector normal del plano empleado, de aucerdo a la posici n del plano de corte, se definir  donde estar  el punto
  #y por consiguiente el s lido generado despu s del corte. El signo de offset determina hacia donde apunta el vector normal, por tanto determina que parte del s lido usar.

  aux_vector=cut_plane.Normal

  if offset<0:
    aux_point=Point.ByCartesianCoordinates(Identity,aux_vector.X,aux_vector.Y,aux_vector.Z)
  else:
    aux_point=Point.ByCartesianCoordinates(Identity,-aux_vector.X,aux_vector.Y,aux_vector.Z)

  sem_elli_sol=elli_sol.Trim(cut_plane,aux_point)
  #Al parece se obtiene una lista por lo tanto se iguala la misma variable al primer elemento de esa lista  ? ...Si funciona :)
  sem_elli_sol=sem_elli_sol[0]

  #Lista que contiene plano de chafl n y punto para trim
  cha_vec=[]
  cha_vec=get_chaplane(rad_x,rad_y,height,offset)

  #Obtenci n de s lido con chafl n
  cha_sol=sem_elli_sol.Trim(cha_vec[0],cha_vec[1])


  return cha_sol

def rec(ancho, largo, altura):


  Identidad=CoordinateSystem.Identity();

  #Creaci n de Sistema Coordenado en punto m s alto

  xc=0;
  yc=0;

  origen=Point.ByCoordinates(xc,yc,altura)


  cs=CoordinateSystem.ByOriginVectors(origen, Identidad.XAxis, Identidad.YAxis,\_
   Identidad.ZAxis)

  #Creaci n de rectangulos

  rec_1=Rectangle.ByWidthLength(Identidad, ancho, largo)

  rec_2=Rectangle.ByWidthLength(cs,ancho, largo)

  rec_vec=[]
  rec_vec.append(rec_1)
  rec_vec.append(rec_2)

  #Creaci n de s lido

  rec_sol=Solid.ByLoft(rec_vec)

  #Libera Geometr as intermedias
  rec_1.Dispose()
  rec_2.Dispose()
  for i in rec_vec:
    i.Dispose()

  return rec_sol

def sem_elli_YZ(rad_x,rad_y,height,offset):
  """
  Descripci n:
    Genera un s lido semiel ptico mediante la generaci n de un elipse
    haciendo un corte con ayuda de un plano YZ, adem s se puede modificar
    la posici n en X del plano para obtener un s lido de distinta geometr a.
    La posici n del s lido se encuentra en el centroide del primer elipse como
    origen, por lo que el semielipse se encuentra partiendo del lado recto en el
    origen coordenado, se incluyo una funci n que permite trasladar al origen dicho
    s lido al cambiar la posici n del plano de corte. Es importante mencionar que el
    double "offset" debe estar contenido en el rango -rad_x < offset < rad_x para que
    funcione de manera adecuada, dependiendo el signo que tome esta variable,
    el elipse cambiar  de cuadrante.

  Argumentos:
    rad_x: radio en direcci n de la coordenada X.
    rad_y: radio en direcci n de la coordenada Y.
    height: n mero que funge como propiedad del solido y que determinar
      la separaci n entre planos.
    offset: n mero que define la posici n del plano de corte en el eje X,
      con rango -rad_x < offset < rad_x

  Variables:
    Identity: Sistema coordenado en el origen.
    xc: Variable usada para definir la coordenada X para el sistema coordenado
      en el punto m s alto.
    yc: Variable usada para definir la coordenada Y para el sistema coordenado
      en el punto m s alto.
    tall_pt: tallEST_pOINt es un punto utilizado para definir el sistema
      coordenado m s alto.
    cs: cOORDINATEsYSTEM es el sistema coordenado en el punto m s alto del s lido.
```

```
    elli_1: elliPSE_1 curva inferior en forma de elipse utilizada para generar
      el s lido
    elli_2: elliPSE_2 curva superior en forma de elipse utilizada para generar
      el s lido
    elli_vec: elliPSE_vecTOR lista que contiene las dos curvas de elipses para
      generar el s lido.
    elli_sol: elliPSE_solID S lido con forma de elipse completo.
    cut_plane: Plano de corte de elipse.
    aux_vector: vector auxiliar normal para generar el cortado del s lido.
    aux_point: Punto necesario para realizar la operaci n del trim, se obtiene
      este punto para conocer que secci n es la que se necesita.

"""

#Creaci n de Sistema Coordenado en el origen

Identity=CoordinateSystem.Identity();

#Creaci n de Sistema Coordenado en punto m s alto

xc=0;
yc=0;

tall_pt=Point.ByCoordinates(xc,yc, height)


cs=CoordinateSystem.ByOriginVectors( tall_pt, Identity.XAxis, Identity.YAxis, Identity.ZAxis)

#Creaci n de elipses

elli_1=Ellipse.ByCoordinateSystemRadii(Identity, rad_x, rad_y)

elli_2=Ellipse.ByCoordinateSystemRadii(cs, rad_x, rad_y)

elli_vec=[]
elli_vec.append(elli_1)
elli_vec.append(elli_2)

#Creaci n de s lido completo

elli_sol=Solid.ByLoft(elli_vec)

#Traslado de la geometr a al origen (desde python script no puedo trasladar el trim,
#tengo que trasladar la figura antes de cortarla)

#elli_sol=elli_sol.Translate(-offset,0,0)
#No es necesario trasladar porque la torre mayor mediante rectangulo ya tiene un desplazamiento

#Trim del s lido

#Plano de corte, se determina mediante el plano (YZ) y con un offset que puede
#ser desde - rad_x hasta + rad_x

cut_plane=Plane.ByOriginNormal(Point.ByCoordinates(0,0,0),Vector.ByCoordinates(1,0,0));


cut_plane=cut_plane.Offset(offset);

#Punto Auxiliar
#Se busca mediante el vector normal del plano empleado, de aucerdo a la posici n del
#plano de corte, se definir  donde estar  el punto y por consiguiente el s lido
#generado despu s del corte. El signo de offset determina hacia donde apunta
#el vector normal, por tanto determina que parte del s lido usar.

aux_vector=cut_plane.Normal

if offset<0:
  aux_point=Point.ByCartesianCoordinates(Identity,aux_vector.X,aux_vector.Y,aux_vector.Z)
else:
  aux_point=Point.ByCartesianCoordinates(Identity,-aux_vector.X,aux_vector.Y,aux_vector.Z)

sem_elli_sol=elli_sol.Trim(cut_plane,aux_point)

#Libera geometr a intermedia
elli_1.Dispose()
elli_2.Dispose()
elli_sol.Dispose()
cut_plane.Dispose()

return sem_elli_sol

def Torre_Mayor(width, length, height):

#Factores de dimensiones para elipse

rad_x=1.64559725*width
rad_y=0.90034747*length
offset=0.87461398*width
rec_height=0.88078205*height

#Se genera el edificio rectangular
rect=rec(width,length,rec_height)

#Se genera el semielipse de los primeros niveles
first_elli_height=0.98290221*rec_height
first_elli=sem_elli_YZ(rad_x,rad_y,first_elli_height,offset)

#Se genera el trapecio intermedio entre el edificio rectangular y
#el edificio semiel ptico

#Se implement  la funcion que genera un semielipse mediante geometr a anal tica para
#conseguir la distancia real en eje Y de los puntos, ya que al cortar el semielipse,
#las longitudes no son las especificadas

real_y=math.sqrt((1-(offset*offset)/(rad_x*rad_x))*(rad_y*rad_y))

#Se obtienen los puntos de la superficie rectangular del prisma eliptico
elli_pts=[]
```

```
    elli_pts.append(Point.ByCoordinates(offset,real_y,rec_height))
    elli_pts.append(Point.ByCoordinates(offset,-real_y,rec_height))
    elli_pts.append(Point.ByCoordinates(offset,-real_y,0))
    elli_pts.append(Point.ByCoordinates(offset,real_y,0))

    #Se genera el rectangulo mediante los puntos descritos antes
    elli_rec=Rectangle.ByCornerPoints(elli_pts)

    #Se obtienen los puntos de la superficie rectangular que apunta hacia la elipse
    rec_pts=[]
    rec_pts.append(Point.ByCoordinates(width/2,length/2,0))
    rec_pts.append(Point.ByCoordinates(width/2,-length/2,0))
    rec_pts.append(Point.ByCoordinates(width/2,-length/2,rec_height))
    rec_pts.append(Point.ByCoordinates(width/2,length/2,rec_height))

    #Se genera el rectangulo mediante los puntos descritos antes
    rec_rec=Rectangle.ByCornerPoints(rec_pts)

    #Se crea una lista que contenga los rectangulos de las 2 superficies
    rects=[]
    rects.append(elli_rec)
    rects.append(rec_rec)

    #Se genera el s lido con forma de trapecio
    trapeze=Solid.ByLoft(rects)

    #Se genera el segundo elipse, es el que contiene el chafl n
    sec_elli_height=0.15245245*rec_height
    sec_elli=sem_elli_YZ_cha(rad_x,rad_y,sec_elli_height,offset)

    sec_elli=sec_elli[0]
    sec_elli=sec_elli.Translate(offset,0,first_elli_height)

    #Se genera segundo rectangulo, localizado en la azotea
    sec_rec_width=offset-width/2
    sec_rec_length=length
    sec_rec_height=0.1089854*rec_height

    sec_rec=rec(sec_rec_width,sec_rec_length,sec_rec_height)

    sec_rec=sec_rec.Translate((width/2)+sec_rec_width/2,0,rec_height)

    #Se genera tercer elipse, localizada en la azotea

    sec_rad_x=0.8205798*width
    sec_rad_y=0.5*length
    sec_offset=-0.025*width
    third_elli_height=0.1089854*rec_height

    third_elli=sem_elli_YZ(sec_rad_x,sec_rad_y,third_elli_height,sec_offset)
    third_elli=third_elli[0]
    third_elli=third_elli.Translate(-sec_offset+width/2,0,rec_height)

    tower_union=rect.UnionAll([first_elli[0],trapeze,sec_elli,sec_rec,third_elli])

    #Libera Geometr as intermedias
    elli_rec.Dispose()
    rec_rec.Dispose()
    rect.Dispose()
    for i in first_elli:
      i.Dispose()
    trapeze.Dispose()
    sec_elli.Dispose()
    sec_rec.Dispose()
    third_elli.Dispose()

    return tower_union


#Assign your output to the OUT variable

tower_gen=Torre_Mayor(IN[0],IN[1],IN[2])

OUT = tower_gen
```

A.2. Código: Intersección de Niveles

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN

"""///*** Main ***///"""

pol_surf=get_floor_curves(IN[0],IN[1],IN[2])
per_curves=get_floor_per_curves(pol_surf)
pol_curves=get_floor_pol_curves(per_curves)
conec_parameters=[IN[1], IN[2]]

#Libera per_curves
del per_curves

#Assign your output to the OUT variable
OUT = conec_parameters,pol_surf,pol_curves


"""///*** Functions ***///"""


def get_floor_curves(solid,height,n_floors):
  """
  Descripci n:
    Genera una lista con datos de superficies de Dynamo, que son los pisos del edificio idealizados como planos, la forma de generarlos es mediante el numero de pisos deseado y la altura que tenga el s lido, el problema de esto es que tiene la misma distancia entre planos pero sta puede ser de tipo flotante y poco pr ctica.

  Argumentos:
    solid: s lido rectangular al que se le har el proceso de extracci n de planos.
    height: n mero que funge como propiedad del solido y que determinar la separaci n entre planos.
    n_floors: entero que define el numero de pisos contenidos en el edificio o solido rectangular

  Variables:
    center: Variable que define el centroide del s lido
    floor_dist: floor_distANCE define la separaci n de los pisos mediante la altura y el numero de pisos deseado (recordar que debido a este procedimiento pueden resultar separaciones con punto flotante y no pr cticas)
    floor_height: Lista que contiene las alturas de cada entrepiso
    floor_cent: floor_centROID contiente los centroides de cada piso
    floor_planes: Contiene los planos idealizados de cada piso, con el vector normal siempre apuntando hacia el eje vertical
    floor_inter: floor_interSECT Contiene las superficies generadas por cada corte de cada piso

  """

  center=solid.Centroid()


  floor_dist=height/n_floors
  floor_height=[]

  #Recordar que no se va a ajustar como la funci n de dynamoscript sino que tendr un error para separaciones flotantes
  i=0
  while i<n_floors:
    aux=i*floor_dist
    floor_height.append(aux)
    i=i+1
  floor_height.append(height)


  roof_proof=height-floor_height[n_floors-1]
  n_floors=n_floors+1
  #Esto es para evitar tener un ltimo piso m s peque o de lo solicitado y mejor "ahorrarnos" espacio brindandole al ltimo nivel una altura mayor, si as sucediera.
  #if roof_proof<floor_dist:
  # floor_height.pop(n_floors-1)


  #Listas necesarias
  floor_cent=[]
  floor_planes=[]
  floor_inter=[]
  floor_inter_aux=[]
  inter_pisos_curvas=[]


  for i in floor_height:
    floor_cent.append(Point.ByCoordinates(center.X,center.Y,i))


  for i in floor_cent:
    floor_planes.append(Plane.ByOriginNormal(i,Vector.ByCoordinates(0,0,1)))


  for i in floor_planes:
    floor_inter.append(solid.Intersect(i))

  floor_proof=[]
  for i in floor_inter:
    floor_proof.append(PolySurface.ByJoinedSurfaces(i))

  #Elimina matriz y convierte en un elemento de 1D (tipo Flatten)
  #Se omiti por el momento debido a que se implement como prueba la union de surfaces con ByJoinedSurfaces
  """
  for i in range(n_floors):
    for j in range(floor_inter[i].Count):
      aux2=floor_inter[i][j]
      #La siguiente condicional es para evitar tener pisos que no sean representativos y por lo tanto para que la funci n de obteneci n de puntos funcione
      if aux2.Area>0.1:
        floor_inter_aux.append(aux2)
```

```
    """
    #inter_pisos_1.append(inter_pisos_1[i].PerimeterCurves)

    #Libera floor_planes
    for i in floor_planes:
      i.Dispose()

    #Libera floor_inter
    for i in floor_inter:
      for j in i:
        j.Dispose()


    return floor_proof

def get_floor_per_curves(poly_surf):

    peri_curves=[]
    for i in range(poly_surf.Count):
      peri_curves.append(poly_surf[i].PerimeterCurves())

    return peri_curves

def get_floor_pol_curves(peri_curves):


    poly_curves=[]
    for i in range(peri_curves.Count):
      ini_curves=[]
      rest_curves=[]
      ini_curves.append(peri_curves[i][0])
      for j in range(peri_curves[i].Count-1):
        rest_curves.append(peri_curves[i][j+1])

      aux_poly=ini_curves[0].Join(rest_curves)
      poly_curves.append(aux_poly)

    #Libera ini_curves
    for i in ini_curves:
      i.Dispose()

    #Libera rest_curves
    for i in rest_curves:
      i.Dispose()

    return poly_curves
```

A.3. Código: Retícula de Estructuración

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
from System.Threading import Thread, ThreadStart
import time
#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN


"""///*** Main ***///"""

ret=[] #tama o de ret cula
ret_lines_ground=[]

centroids_vec=[]
ret_vec=[]
ret_lines_f=[]
n_floors=IN[0][0][1]
height=IN[0][0][0]
spa=height/n_floors
polysurfaces=IN[0][1]
polycurves=IN[0][2]
sep_ret_vec=IN[2]

local1_time_fin=[]
local2_time_fin=[]
local3_time_fin=[]

global_time_ini=time.time()

fun1_time_ini=time.time()
ret_lines_ground=get_ret_g(IN[1], sep_ret_vec)
for f in range(n_floors+1):
  ret=get_ret(IN[1],spa*f,sep_ret_vec, ret_lines_ground[0])
  ret_vec.append(ret)
ret_lines_f.append(ret_lines_ground[0])
ret_lines_f.append(ret[3])
fun1_time_fin=time.time() - fun1_time_ini


for i in range(ret_lines_ground[1].Count):
  f_point=Point.ByCoordinates(ret_lines_ground[1][i].X, ret_lines_ground[1][i].Y,height)
  ret_lines_f.append(Line.ByStartPointEndPoint(ret_lines_ground[1][i],f_point))

#Creaci n de superficies

ps_1=Point.ByCoordinates(ret_lines_ground[1][0].X, ret_lines_ground[1][0].Y,height)
ps_2=Point.ByCoordinates(ret_lines_ground[1][sep_ret_vec[1]].X,ret_lines_ground[1][sep_ret_vec[1]].Y,height)
ps_3=Point.ByCoordinates(ret_lines_ground[1][sep_ret_vec[1]+1].X,ret_lines_ground[1][sep_ret_vec[1]+1].Y,height)
ps_4=Point.ByCoordinates(ret_lines_ground[1][sep_ret_vec[1]*2+1].X,ret_lines_ground[1][sep_ret_vec[1]*2+1].Y,height)
surf_1=Surface.ByPerimeterPoints([ret_lines_ground[1][0],ps_1,ps_2,ret_lines_ground[1][sep_ret_vec[1]]])
surf_2=Surface.ByPerimeterPoints([ret_lines_ground[1][0],ps_1,ps_3,ret_lines_ground[1][sep_ret_vec[1]+1]])
surf_3=Surface.ByPerimeterPoints([ret_lines_ground[1][sep_ret_vec[1]+1],ps_3,ps_4,ret_lines_ground[1][sep_ret_vec[1]*2+1]])
surf_4=Surface.ByPerimeterPoints([ret_lines_ground[1][sep_ret_vec[1]*2+1],ps_4,ps_2,ret_lines_ground[1][sep_ret_vec[1]]])

per_surf_vec=[surf_1, surf_2, surf_3, surf_4]


global_time_fin=time.time() - global_time_ini
times=[global_time_fin]

#Assign your output to the OUT variable
conec_parameters=[ret_vec,sep_ret_vec, height, n_floors, polysurfaces, polycurves]
#ret_lines_f y per_surf_vec son para visualizaci n de caja verde
OUT = conec_parameters,ret,ret_lines_f, per_surf_vec,times


"""///*** Functions ***///"""


def get_ret_g(ret_dim,sep_in):

  #Asginaci n de un tama o de reticula inicial fijo (Una planta equivale a las distancias de un cuarto de planta real)
  x_max_ret=ret_dim[0]
  y_max_ret=ret_dim[1]

  sep_x=x_max_ret*2/sep_in[0]
  sep_y=y_max_ret*2/sep_in[1]


  ret_nodes_g=[]
  ret_lines=[]
  ret_lines_g=[]

  for i in range(sep_in[0]+1):
    for j in range(sep_in[1]+1):
      aux_point_g=Point.ByCoordinates(-x_max_ret+i*sep_x,-y_max_ret+j*sep_y,0.0)
      ret_nodes_g.append([aux_point_g, aux_point_g.X,aux_point_g.Y])



  #Libera ret_nodes_g
  #del ret_nodes_g

  ret_nodes_y_al_g=[]
  ret_nodes_sort=sorted(ret_nodes_g, key=lambda tup: tup[2])#y_align_nodes_bubble(ret_nodes_y_g,0,0)

  for i in ret_nodes_sort:
    ret_nodes_y_al_g.append(i[0])

  ret_nodes_x_al_g=[]
  ret_nodes_sort=sorted(ret_nodes_g, key=lambda tup: tup[1])#x_align_nodes_bubble(ret_nodes_x_g,0,0)

  for i in ret_nodes_sort:
```

```
    ret_nodes_x_al_g.append(i[0])

  box_points=[]
  for i in range(sep_in[1]+1):
    box_points.append(ret_nodes_x_al_g[
      i])

  for i in range(ret_nodes_x_al_g.Count
    -sep_in[1]-1,ret_nodes_x_al_g.Count
    ):
    box_points.append(ret_nodes_x_al_g[
      i])

  for i in range(1,sep_in[0]):
    box_points.append(ret_nodes_y_al_g[
      i])

  for i in range(ret_nodes_x_al_g.Count
    -sep_in[0],ret_nodes_y_al_g.Count
    -1):
    box_points.append(ret_nodes_y_al_g[
      i])



  #Vigas con direcci n X
  for i in range(ret_nodes_x_al_g.Count
    -1):

    if round(ret_nodes_x_al_g[i].X,8)
    == round(ret_nodes_x_al_g[i+1].X,8)
      \
      and round(ret_nodes_x_al_g[i].Z
    ,8) == round(ret_nodes_x_al_g[i+1].
    Z,8):
      ret_lines_g.append(Line.
    ByStartPointEndPoint(
    ret_nodes_x_al_g[i],
    ret_nodes_x_al_g[i+1]))


  #Vigas con direcci n Y
  for i in range(ret_nodes_y_al_g.Count
    -1):

    if round(ret_nodes_y_al_g[i].Y,8)
    == round(ret_nodes_y_al_g[i+1].Y,8)
      \
      and round(ret_nodes_y_al_g[i].Z
    ,8) == round(ret_nodes_y_al_g[i+1].
    Z,8):
      ret_lines_g.append(Line.
    ByStartPointEndPoint(
    ret_nodes_y_al_g[i],
    ret_nodes_y_al_g[i+1]))


  return ret_lines_g, box_points

def get_ret(ret_dim,height,sep_in,
    ret_lines_g):

  #Asginaci n de un tama o de
    reticula inicial fijo (Una planta
    equivale a las distancias de un
    cuarto de planta real)
  x_max_ret=ret_dim[0]
  y_max_ret=ret_dim[1]

  sep_x=x_max_ret*2/sep_in[0]
  sep_y=y_max_ret*2/sep_in[1]

  ret_nodes=[]
  ret_nodes_0=[]
  ret_lines=[]



  for i in range(sep_in[0]+1):
    for j in range(sep_in[1]+1):
      aux_point=Point.ByCoordinates(-
    x_max_ret+i*sep_x,-y_max_ret+j*
    sep_y,height)
      ret_nodes.append([aux_point,
    aux_point.X,aux_point.Y])
      ret_nodes_0.append(aux_point)




  ret_nodes_y_al=[]
  ret_nodes_sort=sorted(ret_nodes, key=
    lambda tup: tup[2])#
    y_align_nodes_bubble(ret_nodes_y
    ,0,0)

  for i in ret_nodes_sort:
      ret_nodes_y_al.append(i[0])

  ret_nodes_x_al=[]
  ret_nodes_sort=sorted(ret_nodes, key=
    lambda tup: tup[1])#
    x_align_nodes_bubble(ret_nodes_x
    ,0,0)

  for i in ret_nodes_sort:
      ret_nodes_x_al.append(i[0])



  #Vigas con direcci n X
  for i in range(ret_nodes_x_al.Count
    -1):

    if round(ret_nodes_x_al[i].X,8) ==
    round(ret_nodes_x_al[i+1].X,8):
      ret_lines.append(Line.
    ByStartPointEndPoint(ret_nodes_x_al
    [i],ret_nodes_x_al[i+1]))


  #Vigas con direcci n Y
  for i in range(ret_nodes_y_al.Count
    -1):

    if round(ret_nodes_y_al[i].Y,8) ==
    round(ret_nodes_y_al[i+1].Y,8):
      ret_lines.append(Line.
    ByStartPointEndPoint(ret_nodes_y_al
    [i], ret_nodes_y_al[i+1]))



  ret=[x_max_ret,y_max_ret,ret_nodes_0,
    ret_lines,ret_lines_g]



  return ret

def y_align_nodes_bubble(nodes, first_f
    , final_f):
  ordered_vec_f=[]

  #bubble sort con .Y
  for f in range(first_f,final_f+1):
    for i in range(nodes[f-first_f].
    Count):
      for j in range(i,nodes[f-first_f
    ].Count):

        if nodes[f-first_f][i].Y>nodes[
    f-first_f][j].Y and round(nodes[f-
    first_f][i].Z,8) == round(nodes[f-
    first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-
    first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
```

```
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].Y,8)==round(nodes[f-first_f][j].Y,8):
          counter=counter+1
          aux_order_vec.append(nodes[f-first_f][j])
        j=j+1

      for k in range(aux_order_vec.Count):
        for l in range(k,aux_order_vec.Count):
          if aux_order_vec[k].X>aux_order_vec[l].X:
            aux= aux_order_vec[l]
            aux_order_vec[l]=aux_order_vec[k]
            aux_order_vec[k]=aux

      ordered_vec.append(aux_order_vec)
      i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
      for j in range(ordered_vec[i].Count):
        ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f

def x_align_nodes_bubble(nodes, first_f, final_f):
  ordered_vec_f=[]

  #bubble sort con .X
  for f in range(first_f,final_f+1):
    for i in range(nodes[f-first_f].Count):
      for j in range(i,nodes[f-first_f].Count):

        if nodes[f-first_f][i].X>nodes[f-first_f][j].X and round(nodes[f-first_f][i].Z,8) == round(nodes[f-first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].X,8)==round(nodes[f-first_f][j].X,8):
          counter=counter+1
          aux_order_vec.append(nodes[f-first_f][j])
        j=j+1

      for k in range(aux_order_vec.Count):
        for l in range(k,aux_order_vec.Count):
          if aux_order_vec[k].Y>aux_order_vec[l].Y:
            aux= aux_order_vec[l]
            aux_order_vec[l]=aux_order_vec[k]
            aux_order_vec[k]=aux

      ordered_vec.append(aux_order_vec)
      i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
      for j in range(ordered_vec[i].Count):
        ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f



def get_slab_ret(ret_lines, sep, floor_surf):

  """
  Descripci n:
    A partir de las lineas generadas por el m dulo de ret cula, se crean grupos de l neas
    perimetrales a losas, posterior se hace la intersecci n de la superficie total de la
    planta con respecto de las losas y se obtiene un vector de las losas que comprenden la
    planta. Recordar que en algunos casos las losas son mas amplias que el perimetro formado
    por las lineas.
  """
  sep_x=sep[0]
  sep_y=sep[1]
  surfs_lines=[]
  floor_surf_vec=[]
  floor_surf_vec_flat=[]

  ##Tiempo ini
  local1_time_ini=time.time()

  for i in range(sep_x*sep_y):

    ll=ret_lines[i]
    lr=ret_lines[i+sep_y]

    lines_vec=[ll, lr]
    surfs_lines.append(lines_vec)

  ##Tiempo fin
  local1_time_fin.append(time.time() - local1_time_ini)

  ##Tiempo ini
  local2_time_ini=time.time()
  j=0
  k=0
  counter_ini=sep_x*sep_y+sep_y
  while(counter_ini< (ret_lines.Count-sep_x)):

    li=ret_lines[counter_ini]
    ls=ret_lines[counter_ini+sep_x]



    surfs_lines[j].append(li)
    surfs_lines[j].append(ls)
    j=j+1


```

```
    counter_ini=counter_ini+sep_x

    if counter_ini>(ret_lines.Count-sep_x-1):
      k=k+1

    if counter_ini>(ret_lines.Count-sep_x-1) and j<sep_x*sep_y:
      counter_ini=sep_x*sep_y+sep_y+k
##Tiempo fin
local2_time_fin.append(time.time() - local2_time_ini)


##Tiempo ini
local3_time_ini=time.time()

local3_time_ini=time.time()

len1 = int(round(len(surfs_lines)-1 ) ) / 4 + 1

surfs_lines_chop=[surfs_lines[x:x+len1] for x in xrange(0, len(surfs_lines)-1, len1)]

surfs_lines_aux_1=[]
surfs_lines_aux_2=[]
surfs_lines_aux_3=[]
surfs_lines_aux_4=[]


def threaded_ps0():
  for i in surfs_lines_chop[0]:
    aux_curve_1=PolyCurve.ByJoinedCurves(i)
    if "PolyCurve" in str(aux_curve_1.GetType()):
      surfs_lines_aux_1.append(Surface.ByPatch(aux_curve_1))
  for i in surfs_lines_aux_1:
    floor_surf_vec.append(floor_surf.Intersect(i))

  return surfs_lines_aux_1
def threaded_ps1():
  for i in surfs_lines_chop[1]:
    aux_curve_2=PolyCurve.ByJoinedCurves(i)
    if "PolyCurve" in str(aux_curve_2.GetType()):
      surfs_lines_aux_2.append(Surface.ByPatch(aux_curve_2))
  for i in surfs_lines_aux_2:
    floor_surf_vec.append(floor_surf.Intersect(i))

  return surfs_lines_aux_2
def threaded_ps2():
  for i in surfs_lines_chop[2]:
    aux_curve_3=PolyCurve.ByJoinedCurves(i)
    if "PolyCurve" in str(aux_curve_3.GetType()):
      surfs_lines_aux_3.append(Surface.ByPatch(aux_curve_3))
  for i in surfs_lines_aux_3:
    floor_surf_vec.append(floor_surf.Intersect(i))

  return surfs_lines_aux_3
def threaded_ps3():
  for i in surfs_lines_chop[3]:
    aux_curve_4=PolyCurve.ByJoinedCurves(i)
    if "PolyCurve" in str(aux_curve_4.GetType()):
      surfs_lines_aux_4.append(Surface.ByPatch(aux_curve_4))
  for i in surfs_lines_aux_4:
    floor_surf_vec.append(floor_surf.Intersect(i))

  return surfs_lines_aux_4



## Algoritmo de paralelizaci n en m s de 10 sub losas

if surfs_lines.Count>10:
  if surfs_lines_chop.Count == 3:
    threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ))

  else:
    threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ),Thread(ThreadStart(threaded_ps3) ) )

  for t in threads: t.Start()
  for t in threads: t.Join()

  """
  for i in range(surfs_lines_aux.Count):
    if surfs_lines_aux[i] is not None:
      floor_surf_vec.append(surfs_lines_aux)

  """

else:

  surfs_vec=[]

  for i in range(surfs_lines.Count):
    aux_curves=PolyCurve.ByJoinedCurves(surfs_lines[i])
    surfs_vec=Surface.ByPatch(aux_curves)
    floor_surf_vec.append(floor_surf.Intersect(surfs_vec))


  #Libera surfs_vec
  del surfs_vec


for i in range(floor_surf_vec.Count):
  for j in range(floor_surf_vec[i].Count):
    if floor_surf_vec[i] is not None:
      floor_surf_vec_flat.append(floor_surf_vec[i][j])

##Tiempo fin
local3_time_fin.append(time.time() - local3_time_ini)

return floor_surf_vec_flat


def get_solids(surf_vec_sol):
  solids_vec=[]
  centroid_v=[]
  for i in range(surf_vec_sol.Count):
    solids_vec.append(surf_vec_sol[i].Thicken(1, True))
    centroid_v.append(solids_vec[i].Centroid())


  #Libera solidos
  for i in solids_vec:
    i.Dispose()

  return centroid_v


def list_filter(list, filter):
  """
  Descripci n:
```

```
478     Funci n auxiliar que ayuda a
        limpiar listas que contengan
        elementos no deseados,
479     como por ejemplo cuando se crean
        intersecciones entre geometr as y
        se desea ob-
480     tener puntos, muchas veces se crean
          otros elementos como l neas o "
        emptylist"
481     que nos dificultan para un proceso
        subsiguiente, con esta funci n se
        eliminan
482     y nos devuelve una lista con el
        mismo tipo de elemento.
483
484   Argumentos:
485     list: Lista a limpiar
486     filter: Cadena de texto con el
        nombre del elemento a conservar en
        la lista usando
487       una funci n denominada GetType()
        .
488
489   Variables:
490     list_filtered: Lista filtrada final
        .
491
492   """
493   list_filtered=[]
494   for l in list:
495     if filter in str(l.GetType()):
496       list_filtered.append(l)
497   return list_filtered
```

A.4. Código: Obtención de Superficies a Partir de la Retícula

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
from System.Threading import Thread, ThreadStart
import time
#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN

"""///*** Main ***///"""

ret=IN[0][1] #tama o de ret cula
surf_vec=[]
centroids_vec=[]
ret_vec=IN[0][0][0]
ret_lines_f=[]
n_floors=IN[0][0][3]
height=IN[0][0][2]
spa=height/n_floors
polysurfaces=IN[0][0][4]
polycurves=IN[0][0][5]
sep_ret_vec=IN[0][0][1]

local1_time_fin=[]
local2_time_fin=[]
local3_time_fin=[]

global_time_ini=time.time()




fun2_time_ini=time.time()
for f in range(n_floors+1):
  slabs_surf=get_slab_ret(ret_vec[f][3],sep_ret_vec, polysurfaces[f])
  slabs_surf=list_filter(slabs_surf, "Surface")
  surf_vec.append(slabs_surf)
fun2_time_fin=time.time() - fun2_time_ini


fun3_time_ini=time.time()

len1 = int(round(len(surf_vec)-1 ) ) / 4 + 1

surf_vec_chop=[surf_vec[x:x+len1] for x in xrange(0, len(surf_vec)-1, len1)]

surf_vec_aux=[None,None,None,None]

def threaded_ps0():
  for i in surf_vec_chop[0]:
    solids_cent_1=get_solids(i)
    centroids_vec.append(solids_cent_1)
  return centroids_vec
def threaded_ps1():
  for i in surf_vec_chop[1]:
    solids_cent_2=get_solids(i)
    centroids_vec.append(solids_cent_2)
  return centroids_vec
def threaded_ps2():
  for i in surf_vec_chop[2]:
    solids_cent_3=get_solids(i)
    centroids_vec.append(solids_cent_3)
  return centroids_vec
def threaded_ps3():
  for i in surf_vec_chop[3]:
    solids_cent_4=get_solids(i)
    centroids_vec.append(solids_cent_4)
  return centroids_vec


## Algoritmo de paralelizaci n en m s de 10 pisos
if n_floors>10:
  if surf_vec_chop.Count == 3:
    threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ))

  else:
    threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ),Thread(ThreadStart(threaded_ps3) ) )

  for t in threads: t.Start()
  for t in threads: t.Join()

  aux_point_parameter=[]

  for i in range(centroids_vec.Count):
    p_aux=centroids_vec[i][0]
    aux_point_parameter.append([centroids_vec[i],p_aux.Z])

  centroids_vec_pre = sorted(aux_point_parameter, key=lambda tup: tup[1])
  centroids_vec=[]
  for i in centroids_vec_pre:
    centroids_vec.append(i[0])

else:

  for f in range(n_floors+1):
    solids_cent=get_solids(surf_vec[f])
    centroids_vec.append(solids_cent)


##Sort centroides
centroids_vec_aligned=[]
surf_vec_aligned=[]
centroids_vec_x_pre=[]

surface_centroid_sort=[]

for f in range(n_floors+1):

  surface_centroid_sort.append(zip(centroids_vec[f],surf_vec[f]))

for f in range(n_floors+1):

  #Acomodo respecto a X global (Valores de X, desacomodados respecto a Y)
  centroids_vec_x_pre=[]
  centroids_vec_x_pre=sorted(surface_centroid_sort[f], key=lambda vec: vec[0].X)


  #Acomodo respecto a Y usando vectores intermedios

  sort_chop_p=centroids_vec_x_pre[0]
  sort_chop=[]
  sort_chop_pre=[]
  sort_chop_cent=[]
  sort_chop_surf=[]
  for i in range(centroids_vec_x_pre.Count):

    #Si tienen el mismo X, se guarda en un vector que se organizara
    #respecto a Y posteriormente
```

```
    if round(sort_chop_p[0].X, 4)==round(centroids_vec_x_pre[i][0].X,4):
      sort_chop_pre.append(centroids_vec_x_pre[i])

    else:
      sort_chop_pre.sort(key=lambda vec: vec[0].Y)
      for j in range(sort_chop_pre.Count):
        sort_chop_cent.append(sort_chop_pre[j][0])
        sort_chop_surf.append(sort_chop_pre[j][1])
      sort_chop_pre=[]
      sort_chop_p=centroids_vec_x_pre[i]

      sort_chop_pre.append(centroids_vec_x_pre[i])

  #Ultimo vector, como ya no hay un cambio en X proximo, hay que incoporarlo al final

  sort_chop_pre.sort(key=lambda vec: vec[0].Y)
  for j in range(sort_chop_pre.Count):
    sort_chop_cent.append(sort_chop_pre[j][0])
    sort_chop_surf.append(sort_chop_pre[j][1])

  aux_x=[]
  for i in sort_chop_cent:
      aux_x.append(i)

  centroids_vec_aligned.append(aux_x)
  aux_x=[]
  for i in sort_chop_surf:
      aux_x.append(i)

  surf_vec_aligned.append(aux_x)

fun3_time_fin=time.time() - fun3_time_ini


#Libera ret_vec
del ret_vec


global_time_fin=time.time() - global_time_ini
times=[global_time_fin,fun2_time_fin, fun3_time_fin]
#times=[fun3_time_fin,sum(local1_time_fin),sum(local2_time_fin),sum(local3_time_fin)]

#Assign your output to the OUT variable
conec_parameters=[sep_ret_vec, height, n_floors, polysurfaces, polycurves]
#ret_lines_f y per_surf_vec son para visualizaci n de caja verde
OUT = conec_parameters,ret, centroids_vec_aligned, surf_vec_aligned,times


"""///*** Functions ***///"""

def get_ret_g(ret_dim,sep_in):

  #Asginaci n de un tama o de reticula inicial fijo (Una planta equivale a las distancias de un cuarto de planta real)
  x_max_ret=ret_dim[0]
  y_max_ret=ret_dim[1]

  sep_x=x_max_ret*2/sep_in[0]
  sep_y=y_max_ret*2/sep_in[1]


  ret_nodes_g=[]
  ret_lines=[]
  ret_lines_g=[]

  for i in range(sep_in[0]+1):
    for j in range(sep_in[1]+1):
      aux_point_g=Point.ByCoordinates(-x_max_ret+i*sep_x,-y_max_ret+j*sep_y,0.0)
      ret_nodes_g.append([aux_point_g, aux_point_g.X,aux_point_g.Y])



  #Libera ret_nodes_g
  #del ret_nodes_g

  ret_nodes_y_al_g=[]
  ret_nodes_sort=sorted(ret_nodes_g, key=lambda tup: tup[2])#y_align_nodes_bubble(ret_nodes_y_g,0,0)

  for i in ret_nodes_sort:
    ret_nodes_y_al_g.append(i[0])

  ret_nodes_x_al_g=[]
  ret_nodes_sort=sorted(ret_nodes_g, key=lambda tup: tup[1])#x_align_nodes_bubble(ret_nodes_x_g,0,0)

  for i in ret_nodes_sort:
    ret_nodes_x_al_g.append(i[0])

  box_points=[]
  for i in range(sep_in[1]+1):
    box_points.append(ret_nodes_x_al_g[i])

  for i in range(ret_nodes_x_al_g.Count-sep_in[1]-1,ret_nodes_x_al_g.Count):
    box_points.append(ret_nodes_x_al_g[i])

  for i in range(1,sep_in[0]):
    box_points.append(ret_nodes_y_al_g[i])

  for i in range(ret_nodes_x_al_g.Count-sep_in[0],ret_nodes_y_al_g.Count-1):
    box_points.append(ret_nodes_y_al_g[i])



  #Vigas con direcci n X
  for i in range(ret_nodes_x_al_g.Count-1):

    if round(ret_nodes_x_al_g[i].X,8) == round(ret_nodes_x_al_g[i+1].X,8) \
      and round(ret_nodes_x_al_g[i].Z,8) == round(ret_nodes_x_al_g[i+1].Z,8):
      ret_lines_g.append(Line.ByStartPointEndPoint(ret_nodes_x_al_g[i], ret_nodes_x_al_g[i+1]))


  #Vigas con direcci n Y
  for i in range(ret_nodes_y_al_g.Count-1):

    if round(ret_nodes_y_al_g[i].Y,8) == round(ret_nodes_y_al_g[i+1].Y,8) \
```

```
      and round(ret_nodes_y_al_g[i].Z,8) == round(ret_nodes_y_al_g[i+1].Z,8):
        ret_lines_g.append(Line.ByStartPointEndPoint(ret_nodes_y_al_g[i], ret_nodes_y_al_g[i+1]))


  return ret_lines_g, box_points

def get_ret(ret_dim,height,sep_in, ret_lines_g):

  #Asginaci n de un tama o de reticula inicial fijo (Una planta equivale a las distancias de un cuarto de planta real)
  x_max_ret=ret_dim[0]
  y_max_ret=ret_dim[1]

  sep_x=x_max_ret*2/sep_in[0]
  sep_y=y_max_ret*2/sep_in[1]

  ret_nodes=[]
  ret_nodes_0=[]
  ret_lines=[]



  for i in range(sep_in[0]+1):
    for j in range(sep_in[1]+1):
      aux_point=Point.ByCoordinates(-x_max_ret+i*sep_x,-y_max_ret+j*sep_y,height)
      ret_nodes.append([aux_point, aux_point.X,aux_point.Y])
      ret_nodes_0.append(aux_point)




  ret_nodes_y_al=[]
  ret_nodes_sort=sorted(ret_nodes, key=lambda tup: tup[2])#y_align_nodes_bubble(ret_nodes_y,0,0)

  for i in ret_nodes_sort:
      ret_nodes_y_al.append(i[0])

  ret_nodes_x_al=[]
  ret_nodes_sort=sorted(ret_nodes, key=lambda tup: tup[1])#x_align_nodes_bubble(ret_nodes_x,0,0)

  for i in ret_nodes_sort:
      ret_nodes_x_al.append(i[0])






  #Vigas con direcci n X
  for i in range(ret_nodes_x_al.Count-1):

    if round(ret_nodes_x_al[i].X,8) == round(ret_nodes_x_al[i+1].X,8):
      ret_lines.append(Line.ByStartPointEndPoint(ret_nodes_x_al[i],ret_nodes_x_al[i+1]))


  #Vigas con direcci n Y
  for i in range(ret_nodes_y_al.Count-1):

    if round(ret_nodes_y_al[i].Y,8) == round(ret_nodes_y_al[i+1].Y,8):
      ret_lines.append(Line.ByStartPointEndPoint(ret_nodes_y_al[i], ret_nodes_y_al[i+1]))



  ret=[x_max_ret,y_max_ret,ret_nodes_0, ret_lines,ret_lines_g]



  return ret

def y_align_nodes_bubble(nodes, first_f, final_f):
  ordered_vec_f=[]

  #bubble sort con .Y
  for f in range(first_f,final_f+1):
    for i in range(nodes[f-first_f].Count):
      for j in range(i,nodes[f-first_f].Count):

        if nodes[f-first_f][i].Y>nodes[f-first_f][j].Y and round(nodes[f-first_f][i].Z,8) == round(nodes[f-first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].Y,8)==round(nodes[f-first_f][j].Y,8):
          counter=counter+1
          aux_order_vec.append(nodes[f-first_f][j])
        j=j+1

      for k in range(aux_order_vec.Count):
        for l in range(k,aux_order_vec.Count):
          if aux_order_vec[k].X>aux_order_vec[l].X:
            aux= aux_order_vec[l]
            aux_order_vec[l]=aux_order_vec[k]
            aux_order_vec[k]=aux

      ordered_vec.append(aux_order_vec)
      i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
      for j in range(ordered_vec[i].Count):
        ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f

def x_align_nodes_bubble(nodes, first_f, final_f):
  ordered_vec_f=[]

  #bubble sort con .X
  for f in range(first_f,final_f+1):
```

```
    for i in range(nodes[f-first_f].Count):
      for j in range(i,nodes[f-first_f].Count):

        if nodes[f-first_f][i].X>nodes[f-first_f][j].X and round(nodes[f-first_f][i].Z,8) == round(nodes[f-first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].X,8)==round(nodes[f-first_f][j].X,8):
          counter=counter+1
          aux_order_vec.append(nodes[f-first_f][j])
        j=j+1

      for k in range(aux_order_vec.Count):
        for l in range(k,aux_order_vec.Count):
          if aux_order_vec[k].Y>aux_order_vec[l].Y:
            aux= aux_order_vec[l]
            aux_order_vec[l]=aux_order_vec[k]
            aux_order_vec[k]=aux

      ordered_vec.append(aux_order_vec)
      i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
      for j in range(ordered_vec[i].Count):
        ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f



def get_slab_ret(ret_lines, sep, floor_surf):

  """
  Descripci n:
    A partir de las lineas generadas por el m dulo de ret cula, se crean grupos de l neas
    perimetrales a losas, posterior se hace la intersecci n de la superficie total de la
    planta con respecto de las losas y se obtiene un vector de las losas que comprenden la
    planta. Recordar que en algunos casos las losas son mas amplias que el perimetro formado
    por las lineas.
  """
  sep_x=sep[0]
  sep_y=sep[1]
  surfs_lines=[]
  floor_surf_vec=[]
  floor_surf_vec_flat=[]

  ##Tiempo ini
  local1_time_ini=time.time()

  for i in range(sep_x*sep_y):

    ll=ret_lines[i]
    lr=ret_lines[i+sep_y]

    lines_vec=[ll, lr]
    surfs_lines.append(lines_vec)

  ##Tiempo fin
  local1_time_fin.append(time.time() - local1_time_ini)

  ##Tiempo ini
  local2_time_ini=time.time()
  j=0
  k=0
  counter_ini=sep_x*sep_y+sep_y
  while(counter_ini< (ret_lines.Count-sep_x)):

    li=ret_lines[counter_ini]
    ls=ret_lines[counter_ini+sep_x]



    surfs_lines[j].append(li)
    surfs_lines[j].append(ls)
    j=j+1


    counter_ini=counter_ini+sep_x

    if counter_ini>(ret_lines.Count-sep_x-1):
      k=k+1

    if counter_ini>(ret_lines.Count-sep_x-1) and j<sep_x*sep_y:
      counter_ini=sep_x*sep_y+sep_y+k
  ##Tiempo fin
  local2_time_fin.append(time.time() - local2_time_ini)


  ##Tiempo ini
  local3_time_ini=time.time()

  local3_time_ini=time.time()

  len1 = int(round(len(surfs_lines)-1 ) ) / 4 + 1

  surfs_lines_chop=[surfs_lines[x:x+len1] for x in xrange(0, len(surfs_lines)-1, len1)]

  surfs_lines_aux_1=[]
  surfs_lines_aux_2=[]
  surfs_lines_aux_3=[]
  surfs_lines_aux_4=[]


  def threaded_ps0():
    for i in surfs_lines_chop[0]:
      aux_curve_1=PolyCurve.ByJoinedCurves(i)
      if "PolyCurve" in str(aux_curve_1.GetType()):
        surfs_lines_aux_1.append(Surface.ByPatch(aux_curve_1))
    for i in surfs_lines_aux_1:
      floor_surf_vec.append(floor_surf.Intersect(i))

    return surfs_lines_aux_1
  def threaded_ps1():
    for i in surfs_lines_chop[1]:
```

```
      aux_curve_2=PolyCurve.ByJoinedCurves(i)
      if "PolyCurve" in str(aux_curve_2.GetType()):
        surfs_lines_aux_2.append(Surface.ByPatch(aux_curve_2))
    for i in surfs_lines_aux_2:
      floor_surf_vec.append(floor_surf.Intersect(i))

    return surfs_lines_aux_2
  def threaded_ps2():
    for i in surfs_lines_chop[2]:
      aux_curve_3=PolyCurve.ByJoinedCurves(i)
      if "PolyCurve" in str(aux_curve_3.GetType()):
        surfs_lines_aux_3.append(Surface.ByPatch(aux_curve_3))
    for i in surfs_lines_aux_3:
      floor_surf_vec.append(floor_surf.Intersect(i))

    return surfs_lines_aux_3
  def threaded_ps3():
    for i in surfs_lines_chop[3]:
      aux_curve_4=PolyCurve.ByJoinedCurves(i)
      if "PolyCurve" in str(aux_curve_4.GetType()):
        surfs_lines_aux_4.append(Surface.ByPatch(aux_curve_4))
    for i in surfs_lines_aux_4:
      floor_surf_vec.append(floor_surf.Intersect(i))

    return surfs_lines_aux_4



  ## Algoritmo de paralelizaci n en m s de 10 sub losas

  if surfs_lines.Count>10:
    if surfs_lines_chop.Count == 3:
      threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ))

    else:
      threads = (Thread(ThreadStart(threaded_ps0) ),Thread(ThreadStart(threaded_ps1) ), Thread(ThreadStart(threaded_ps2) ),Thread(ThreadStart(threaded_ps3) ) )

    for t in threads: t.Start()
    for t in threads: t.Join()

    """
    for i in range(surfs_lines_aux.Count):
      if surfs_lines_aux[i] is not None:
        floor_surf_vec.append(surfs_lines_aux)

    """

  else:

    surfs_vec=[]

    for i in range(surfs_lines.Count):
      aux_curves=PolyCurve.ByJoinedCurves(surfs_lines[i])
      surfs_vec=Surface.ByPatch(aux_curves)
      floor_surf_vec.append(floor_surf.Intersect(surfs_vec))


    #Libera surfs_vec
    del surfs_vec


  for i in range(floor_surf_vec.Count):
    for j in range(floor_surf_vec[i].Count):
      if floor_surf_vec[i] is not None:
        floor_surf_vec_flat.append(floor_surf_vec[i][j])

  ##Tiempo fin
  local3_time_fin.append(time.time() - local3_time_ini)

  return floor_surf_vec_flat


def get_solids(surf_vec_sol):
  solids_vec=[]
  centroid_v=[]
  for i in range(surf_vec_sol.Count):
    solids_vec.append(surf_vec_sol[i].Thicken(1, True))
    centroid_v.append(solids_vec[i].Centroid())


  #Libera solidos
  for i in solids_vec:
    i.Dispose()

  return centroid_v


def list_filter(list, filter):
  """
  Descripci n:
    Funci n auxiliar que ayuda a limpiar listas que contengan elementos no deseados,
    como por ejemplo cuando se crean intersecciones entre geometr as y se desea ob-
    tener puntos, muchas veces se crean otros elementos como l neas o "emptylist"
    que nos dificultan para un proceso subsiguiente, con esta funci n se eliminan
    y nos devuelve una lista con el mismo tipo de elemento.

  Argumentos:
    list: Lista a limpiar
    filter: Cadena de texto con el nombre del elemento a conservar en la lista usando
      una funci n denominada GetType().

  Variables:
    list_filtered: Lista filtrada final.

  """
  list_filtered=[]
  for l in list:
    if filter in str(l.GetType()):
      list_filtered.append(l)
  return list_filtered
```

A.5. Códigos: Catálogos de Materiales

A.5.1. Código: Concreto

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import sys
sys.path.append(r'C:\Program Files (x86)\IronPython 2.7\Lib')
import os
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

## CONCRETO ##
"""///*** Main ***///"""

#B squeda de Cat logo
file_loc=os.getcwd()+str("\MECA\CatalogoCONCR.dat")
file=open(file_loc,'r')

n_sec=file.readline()

sec_index=IN[0]
i_floor=IN[1]
f_floor=IN[2]

for i in range(int(sec_index)):
  sec_prop=file.readline()

  properties=[]
  properties=sec_prop.split()

  selected=[properties[1], i_floor, f_floor]
  for j in range(int(properties[6])):
    read_aux=file.readline()
file.close()

selected.append([3,"CatalogoCONCR", sec_index,file_loc])
#Assign your output to the OUT variable.
OUT = selected
```

A.5.2. Código: Acero Rolado en Caliente

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import sys
sys.path.append(r'C:\Program Files (x86)\IronPython 2.7\Lib')
import os
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

## Acero Rolado en Caliente ##
"""///*** Main ***///"""

#B squeda de Cat logo
file_loc=os.getcwd()+str("\MECA\CatalogoRC.dat")
file=open(file_loc,'r')

n_sec=file.readline()

sec_index=IN[0]
i_floor=IN[1]
f_floor=IN[2]

for i in range(int(sec_index)):
  sec_prop=file.readline()

  properties=[]
  properties=sec_prop.split()

  selected=[properties[1], i_floor, f_floor]

file.close()

selected.append([2,"CatalogoRC", sec_index])
#Assign your output to the OUT variable.
OUT = selected
```

A.5.3. Código: Acero Rolado en Frío

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import sys
sys.path.append(r'C:\Program Files (x86)\IronPython 2.7\Lib')
import os
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

## Acero Rolado en Fr o ##
"""///*** Main ***///"""

#B squeda de Cat logo
file_loc=os.getcwd()+str("\MECA\CatalogoRF.dat")
file=open(file_loc,'r')


n_sec=file.readline()

sec_index=IN[0]
i_floor=IN[1]
f_floor=IN[2]

for i in range(int(sec_index)):
  sec_prop=file.readline()

  properties=[]
  properties=sec_prop.split()
```

```
29
30   selected=[properties[1], i_floor,
       f_floor]
31
32 file.close()
33
```

```
34 selected.append([2,"CatalogoRF",
       sec_index])
35 #Assign your output to the OUT variable
       .
36 OUT = selected
```

A.6. Código: Generador Estructural

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
from math import sqrt
import math
import time

#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN



"""///*** Main ***///"""

#Variables globales
grid_nodes_floor=[] #Arreglo que guarda los nodos de la ret cula COMPLETA para VIGAS
perimetral_points=[] #Arreglo con puntos exteriores para aplicar funci n de conectividades
beam_nodes=[]
nodes_obj_vec=[]
nodes_ind_vec=[]
bframes_obj_vec=[]
bframes_ind_vec=[]
cframes_obj_vec=[]
cframes_ind_vec=[]

ret_data=IN[0]
sep_ret_vector=ret_data[0][0]
t_height=ret_data[0][1]
t_floors=ret_data[0][2]
f_polysurfaces=ret_data[0][3]
f_polycurves=ret_data[0][4]

local1_time_fin=[]
local2_time_fin=[]
local3_time_fin=[]

class frame:

  def __init__(self, id):
    self.id=id
    self.loads_vec=[]

  def add_frame(self,added_frame):
    self.frame=added_frame

  def add_ini_point(self, added_ini_point):
    self.ini_point=added_ini_point

  def add_fin_point(self, added_fin_point):
    self.fin_point=added_fin_point

  def __del__(self):
    print #Destructor

class node:

  def __init__(self, id):
    self.id=id
    self.point=0

  def add_point(self, added_point):
    self.point=added_point

  def __del__(self):
    print #Destructor


global_time_ini=time.time()

fun1_time_ini=time.time()
#Obtenci n de nodos "recortados" o "filtrados" por superficies en todos los niveles.
nodes_int=get_int_cols_nodes(f_polysurfaces,t_floors,t_height,0,t_floors,f_polycurves,ret_data[1])
fun1_time_fin=time.time() - fun1_time_ini

fun2_time_ini=time.time()
#Obtenci n de vigas
beams_complete=get_t_beams(f_polysurfaces,t_height,sep_ret_vector,0,t_floors,f_polycurves,grid_nodes_floor,ret_data[1])
fun2_time_fin=time.time() - fun2_time_ini

fun3_time_ini=time.time()
#Se obtienen las columnas internas y externas regidas por los puntos iniciales
#y finales de las vigas y que se guardan en el arreglo vigas[0]
columns_int_ext=get_int_cols(0,t_floors,beams_complete[1])


beams_data_in=IN[1]
columns_data_in=IN[2]
material_data=material(beams_data_in,columns_data_in, beams_complete[2], columns_int_ext[1], 0, t_floors)
fun3_time_fin=time.time() - fun3_time_ini

global_time_fin=time.time() - global_time_ini
times=[global_time_fin,fun1_time_fin,fun2_time_fin,fun3_time_fin]
#times=[fun2_time_fin,local1_time_fin,local2_time_fin,local3_time_fin]

elements=[columns_int_ext,beams_complete,material_data,times]
conec_parameters=[ret_data]


#Assign your output to the OUT variable
OUT = conec_parameters,elements



"""///*** Functions ***///"""


# COLUMNAS --------------------
# get_int_cols_nodes -
# (obtencion de nodos internos a la superficie)
# get_int_cols -
# obtencion de columnas internas a partir de nodos
# ------------------------

def get_int_cols_nodes(surfaces, n_floors, height, first_f, final_f, curves, grid):

  """
  Descripcion:
    Obtiene los nodos internos a la superficie del nivel contenido en el solido, nece-
    sita una reticula inicial que de la informacion de coordenadas para
```

```
    poder realizar
120     el recorte segun la extension del
        nivel y su forma.
121
122   Argumentos:
123     surfaces: niveles obtenidos del
        code block con la funcion "
        get_floor_curves" y del
124       tipo PolySurfaces.
125     n_floors: numero de pisos que se
        planea estructurar
126     height: numero que funge como
        propiedad del solido y que
        determinar  la separaci n
127       entre planos, sin embargo es la
        altura total, se debe poner el
        par metro entre
128       los pisos que se estructurar n.
129     spa_in: spaCING_inTERIOR Vector que
         contiene la separaci n de la
        ret cula total
130
131
132     first_f: first_fLOOR dato que
        indica donde comenzar a estructurar
133     final_f: final_fLOOR dato que
        indica donde se termina de
        estructurar
134
135
136     curves: Perimetro de la superficie
        de cada nivel a estructurar del
        tipo PolyCurve,
137       verificar si se puede estructurar
         superficies con huecos.
138     grid: Informacion de la ret cula
        utilizada para definir la
        estructuracion interior,
139       contiene un vector con [0]
        dimension max en X, [1] dimension
        max en Y y [2] li-
140       sta de nodos en ret cula
141
142
143   Variables:
144     spa: spaCING espaciamiento vertical
         entre pisos.
145     x_max_grid: Distancia maxima de
        ret cula TOTAL en direccion X.
146     y_max_grid: Distancia maxima de
        ret cula TOTAL en direccion Y.
147       *Asignacion de un tama o de
        reticula inicial fijo (Una planta
        equivale a las
148         distancias de un cuarto de
        planta real)
149       **Recordar que estos numeros
        deben ser tipo flotantes (incluso
        si son sliders)
150         aunque sean n meros cerrados,
        generan error cuando se ingresan
        como enteros
151     sur_points_para:
        surFACE_points_paraMETER Variable
        utilizada como parametro para
152       funcion Curve.PointAtParameter se
        dejo fijo en 0.25 como prueba de
        comporta-
153       miento este parametro ayudar  a
        encontrar puntos alrededor de la
        superficie.
154     int_nodes_floor:
        intERIOR_nodes_floor Lista que
        guarda otra lista por cada nivel
155       con los puntos dados por la
        reticula, pero que se encuentran en
         el interior de
156       la superficie.
157     sur_points_vec:
        surFACE_points_vecTOR Lista
        utilizada dentro de un ciclo por
        cada
158       piso para guardar los puntos
        contenidos en cada curva por nivel,
159       por ello se
        actualiza en cada iteracion.
160       *el parametro va de 0 a 1 por
        ello se eligi  0.25 para poder
        tener 4 puntos
161         por nivel.
162       ** Aqui se necesitan Policurvas
        para encontrar el perimetro de todo
         el nivel
163         por ello falla cuando existen
        huecos, hay que encontrar la manera
         de usar
164         curvas simples y que haga un
        testeo de los extremos.
165
166     x_max: Variable que guarda el
        l mite maximo en direccion X de
        los puntos de curvas
167         perimetrales por cada nivel para
        realizar comparacion entre nodos de
         reticula.
168     y_max: Variable que guarda el
        l mite m ximo en direcci n Y de
        los puntos de curvas
169         perimetrales por cada nivel para
        realizar comparaci n entre nodos
        de reticula.
170     spa_x: spaCING_x Variable utilizada
         para obtener el espaciamiento en
        direcci n X
171     spa_y: spaCING_y Variable utilizada
         para obtener el espaciamiento en
        direcci n Y
172       *Debido a que las dimensiones
        utilizadas para la ret cula total
        es un cuarto de
173         de tama o de la real se
        utilizaron las variables anteriores
        .
174     int_nodes: intERIOR_nodes Vector
        por cada piso que guarda los nodos
        que se utili-
175         zar n en las columnas interiores
        , las cuales est n dentro de la
        curva perime-
176         tral.
177     aux_point: auxILIAR_point utilizado
         para guardar un punto de la
        ret cula total,
178         el cual mediante un ciclo ser
        comparado con los l mites
        definidos en la
179         curva perimetral.
180     grid_nodes: Vector por cada nivel
        que guarda la ret cula completa,
        la cual ser  ut-
181         ilizado para obtener las vigas
        interiores.
182     int_nodes_trim: intERIOR_nodes_trim
         vector por nivel que resulta del
        recorte o Trim
183         entre los puntos de la ret cula
        y las superficies de cada piso.
184     int_nodes_flatrim:
        intERIOR_nodes_flatTENtrim
        utilizado para eliminar vectores
        vac os
185         debidos a la funci n de trim.
186     aux_list: auxILIAR_list vector por
        nivel que ayuda a ingresar los
        nodos por nivel
187         a la variable int_nodes_floor.
188   """
189 #Separacion en entrepisos
190   spa=height/n_floors
191
192
193   """
194   Parametro para encontrar los extremos
       en superficies utilizando funci n
       que agrega
195   puntos en la curva perimetral.
196   """
197   sur_points_para=0.25
```

```
198
199   #Lista que contiene otras listas por cada nivel para guardar los nodos internos
200   int_nodes_floor=[]
201
202   sep_curves=[]
203
204
205   #final_f mas 1 para considerar la azotea para las vigas
206   for f in range(first_f,final_f+1):
207     """
208     #Busqueda de extremos en curvas perimetrales de cada nivel
209     #Lista que guardaran puntos por cada nivel n=(1/sur_points_para)
210     sur_points_vec=[]
211
212     #Variable para restingir el ciclo
213     i=0.0
214     while i<1:
215       #Funcion que encuentra o divide curvas de acuerdo a un parametro de 0 a 1
216       sur_points_vec.append(curves[f].PointAtParameter(i*sur_points_para))
217       i=sur_points_para+i
218     """
219
220
221     grid_nodes=[] #Arreglo que guarda los nodos de la reticula COMPLETA para VIGAS
222     aux_per=[]
223
224     """
225     Obtencion de curves a partir de polycurves perimetrales, para obtener los puntos
226     iniciales y finales, verificar que no se repitan y agregarlos a los grid_nodes
227     y int_nodes, ya que  stos  son los v rtices de las superficies que antes no se
228     a ad an.
229     """
230
231     aux_c=curves[f-first_f].Curves()
232
233     sep_curves.append(aux_c)
234
235     ext_curve_points=[]
236
237     ext_curve_points.append(sep_curves[f-first_f][0].StartPoint)
238
239     ext_curve_points_flag=0
240
241     for i in range(sep_curves[f-first_f].Count):
242
243       ext_curve_points_flag=0
244       for j in range(ext_curve_points.Count):
245
246         if ext_curve_points[j].IsAlmostEqualTo(sep_curves[f-first_f][i].StartPoint):
247           ext_curve_points_flag=1
248
249       if ext_curve_points_flag==0:
250         ext_curve_points.append(sep_curves[f-first_f][i].StartPoint)
251
252
253     for i in range(sep_curves[f-first_f].Count):
254
255       ext_curve_points_flag=0
256       for j in range(ext_curve_points.Count):
257
258         if ext_curve_points[j].IsAlmostEqualTo(sep_curves[f-first_f][i].EndPoint):
259           ext_curve_points_flag=1
260
261       if ext_curve_points_flag==0:
262         ext_curve_points.append(sep_curves[f-first_f][i].EndPoint)
263
264
265     for i in ext_curve_points:
266       grid_nodes.append(i)
267       aux_per.append(i)
268
269     perimetral_points.append(aux_per)
270
271
272     for i in range(grid[2].Count):
273       #Busqueda para conocer si ya existen esos puntos en el vector grid_nodes
274       #Habr  puntos iguales para el caso de planta coincidente con reticula
275       grid_nodes_flag=0
276       aux_point=Point.ByCoordinates(grid[2][i].X,grid[2][i].Y,f*spa)#-x_max_grid+i*spa_x,-y_max_grid+j*spa_y,f*spa)
277
278       for j in range(grid_nodes.Count):
279         if aux_point.IsAlmostEqualTo(grid_nodes[j]):
280           grid_nodes_flag=1
281
282       if grid_nodes_flag==0:
283         #Se guardan los puntos de la ret cula completa
284         grid_nodes.append(aux_point)
285
286
287     #Nodos de la ret cula completa por nivel
288     grid_nodes_floor.append(grid_nodes)
289     #grid_nodes_floor.append(int_nodes)
290
291
292
293   return  grid_nodes_floor#, ext_curve_points#, grid_nodes_floor, sep_curves, ext_curve_points#, grid_nodes#, int_nodes#,, grid_nodes_floor
294
295 def get_int_cols(first_f, final_f, nodes):
296   """
297   Descripci n:
298     Manda a llamar la funci n get_int_cols_nodes() la cual le proporciona los nodos
299     internos obtenidos del recorte de superficie de todos los niveles, despu s uti-
300     liza un ciclo anidado para ligar los puntos de piso-techo y formar columnas que
301     sean ortogonales al nivel.
302
303   Argumentos:
304     surfaces: niveles obtenidos del code block con la funci n "get_floor_curves" y del
305       tipo PolySurfaces.
306     n_floors: n mero de pisos que se planea estructurar (Aqu  se est poniendo el n m-
307       ero total de pisos, se debe hacer m s bien una resta entre first_f y final_f)
```

```
    height: n mero que funge como propiedad del solido y que determinar  la separaci n
      entre planos, sin embargo es la altura total, se debe poner el par metro entre
      los pisos que se estructurar n.
    spa_in: spaCING_inTERIOR Vector que contiene la separaci n de la ret cula total
      (Checar si es necesario, porque a lo mejor grid ya contiene esa informaci n)
    first_f: first_fLOOR dato que indica donde comenzar a estructurar
    final_f: final_fLOOR dato que indica donde se termina de estructurar
    (recomendable: usar un vector solamente)

    curves: Perimetro de la superficie de cada nivel a estructurar del tipo PolyCurve,
      verificar si se puede estructurar superficies con huecos.

  Variables:
    nodes: Nodos que se obtienen de la funci n get_int_cols_nodes() referentes a los
      recortes de puntos con las superficies de todos los pisos.
    int_cols: intERIOR_colUMNs Lista de listas que guardan las columnas encontradas en
      cada piso.
    aux_list_cols: auxILIAR_list_colUMNs Vector auxiliar que guarda las columnas por
      nivel para posteriormente guardarlas en int_cols.
  """



  #Lista de listas de columnas por niveles
  int_cols=[]

  """
  B queda de todos los nodos obtenidos de get_int_cols_nodes() en todos los niveles,
  para ligar los nodos de piso con los nodos de techo y obtener columnas perpendicu-
  lares a las superficies.
  """
  n_floors=final_f-first_f
  index=bframes_ind_vec[final_f-1][bframes_ind_vec[final_f-1].Count-1]+1


  for i in range(n_floors):

    aux_list_cols=[]
    for j in range(nodes[i].Count):
      for k in range(nodes[i+1].Count):

        if round(nodes[i][j].X,3) == round(nodes[i+1][k].X,3) and round(nodes[i][j].Y,3) == round(nodes[i+1][k].Y,3):

          aux_list_cols.append(Line.ByStartPointEndPoint(nodes[i][j], nodes[i+1][k]))

    int_cols.append(aux_list_cols)


```

```
    #Agregado de objetos frame
    aux_fobj_list=[]
    #Lista de indices
    aux_find_obj_list=[]

    for j in range(int_cols[i].Count):

      aux_fobj_list.append(frame(index))
      aux_fobj_list[j].add_frame(int_cols[i][j])
      #Obtenci n de  ndices  para frames
      aux_find_obj_list.append(aux_fobj_list[j].id)

      index=index+1


    #Lista de objetos frame
    cframes_obj_vec.append(aux_fobj_list)
    #Lista de indices de objeto frame
    cframes_ind_vec.append(aux_find_obj_list)

  ##Se agrega una copia del ultimo piso para poder utilizar la funci n search_nodes_frames
  #Lista de objetos frame
  cframes_obj_vec.append(aux_fobj_list)
  #Lista de indices de objeto frame
  cframes_ind_vec.append(aux_find_obj_list)

  node_col_index=search_nodes_frames(cframes_ind_vec,cframes_obj_vec,nodes_ind_vec,nodes_obj_vec)

  ##Se elimina la copia del ultimo piso utilizada por la funci n search_nodes_frames
  #Lista de objetos frame
  cframes_obj_vec.pop(cframes_obj_vec.Count-1)
  #Lista de indices de objeto frame
  cframes_ind_vec.pop(cframes_ind_vec.Count-1)
  #Lista de indices
  node_col_index.pop(node_col_index.Count-1)


  return nodes,cframes_ind_vec,int_cols, node_col_index




# VIGAS - - - - - - - - - - - - - - - - - - - - -
# get_int_beam_nodes --
# obtencion de nodos internos a la superfice
# get_t_beams     --
# obtenci n de vigas internas a partir de nodos
# - - - - - - - - - - - - - - - - - - - - - - - - -


def get_int_beam_nodes(beams, first_f, final_f):

  int_b_nodes_floor=[]

  n_floors=final_f-first_f
  for f in range(n_floors):

    int_b_nodes=[]


    int_b_nodes.append(beams[f][0].StartPoint)
```

```
417     int_b_nodes.append(beams[f][0].
        EndPoint)
418
419
420     for i in range(1,beams[f].Count):
421
422       x_flag=0
423       y_flag=0
424
425       for j in range(int_b_nodes.Count)
        :
426
427         if round(beams[f][i].StartPoint
        .X,6) == round(int_b_nodes[j].X,6)
        and round(beams[f][i].StartPoint.Y
        ,6) == round(int_b_nodes[j].Y,6):
428           x_flag=1
429
430         if round(beams[f][i].EndPoint.X
        ,6) == round(int_b_nodes[j].X,6)
        and round(beams[f][i].EndPoint.Y,6)
         == round(int_b_nodes[j].Y,6):
431           y_flag=1
432
433       if x_flag==0:
434         int_b_nodes.append(beams[f][i].
        StartPoint)
435
436
437       if y_flag==0:
438         int_b_nodes.append(beams[f][i].
        EndPoint)
439
440     int_b_nodes_floor.append(
        int_b_nodes)
441
442
443
444   return int_b_nodes_floor
445
446
447
448
449 def get_t_beams(surfaces, height,
        spa_in, first_f, final_f, curves,
        grid_nodes_floor,ret):
450
451   """
452   Descripci n:
453     Utiliza un lista de listas global (
        grid_nodes_floor) que se obtiene de
         la funci n
454     get_int_cols_nodes() que guarda los
         nodos de la ret cula total, por
        cada nivel,
455     se ciclar  un procedimiento que
        verifica que que el nodo est
        dentro de una zona*
456     y si es as  crea l neas con
        direcci n en X (como se acomod
        el vector global)**
457     despu s se usa un algoritmo de
        ordenamiento para cambiar de orden
        el vector de
458     los nodos reticulares por piso, y
        se vuelve a ciclar para verificar
        que el nodo
459     est  dentro de la zona** pero
        ahora se crean vigas en direcci n
        Y.
460     Se hace un recorte de l nes con
        superficies y se filtra el vector
        final.
461     Posteriormente se hace una llamada
        de funci n get_t_beams_nodes()
        para obtener
462     los nodos de las vigas.
463
464
465     *(por ahora la zona est  dada por
        4 puntos por tanto sirva para
        rectangular)
466     **evita que se generen vigas
        externas para el caso rectangular (
        por ende todos)
467
468
469   Argumentos:
470     surfaces: niveles obtenidos del
        code block con la funci n "
        get_floor_curves" y del
        tipo PolySurfaces.
471
472
473     height: n mero que funge como
        propiedad del solido y que
        determinar  la separaci n
474       entre planos, sin embargo es la
        altura total, se debe poner el
        par metro entre
475       los pisos que se estructurar n.
476     spa_in: spaCING_inTERIOR Vector que
         contiene la separaci n de la
        ret cula total
477       (Checar si es necesario, porque a
         lo mejor grid ya contiene esa
        informaci n)
478
479     first_f: first_fLOOR dato que
        indica donde comenzar a estructurar
480     final_f: final_fLOOR dato que
        indica donde se termina de
        estructurar
481     (recomendable: usar un vector
        solamente)
482
483     curves: Perimetro de la superficie
        de cada nivel a estructurar del
        tipo PolyCurve,
484       verificar si se puede estructurar
         superficies con huecos.
485     grid: Informaci n de la ret cula
        utilizada para definir la
        estructuraci n interior,
486       contiene un vector con [0]
        dimensi n max en X, [1] dimensi n
        max en Y y [2] li-
487       sta de nodos en ret cula
488
489
490   Variables:
491     spa: spaCING espaciamiento vertical
         entre pisos.
492     x_max_grid: Distancia m xima de
        ret cula TOTAL en direcci n X.
493     y_max_grid: Distancia m xima de
        ret cula TOTAL en direcci n Y.
494       *Asignaci n de un tama o de
        reticula inicial fijo (Una planta
        equivale a las
495         distancias de un cuarto de
        planta real)
496       **Recordar que estos numeros
        deben ser tipo flotantes (incluso
        si son sliders)
497         aunque sean n meros cerrados,
        generan error cuando se ingresan
        como enteros
498       ***Ser a mejor encontrar la
        distancia m xima de las
        superficies!!!
499     sur_points_para:
        surFACE_points_paraMETER Variable
        utilizada como parametro para
500       funci n Curve.PointAtParameter
        se dej  fijo en 0.25 como prueba
        de comporta-
501       miento este parametro ayudar  a
        encontrar puntos alrededor de la
        superficie.
502     int_nodes_floor:
        intERIOR_nodes_floor Lista que
        guarda otra lista por cada nivel
503       con los puntos dados por la
        reticula, pero que se encuentran en
         el interior de
504       la superficie.
505     sur_points_vec:
        surFACE_points_vecTOR Lista
```

```
    utilizada dentro de un ciclo por cada
      piso para guardar los puntos contenidos en cada curva por nivel, por ello se
      actualiza en cada iteraci n.
      *el parametro va de 0 a 1 por ello se eligi  0.25 para poder tener 4 puntos
       por nivel.
      ** Aqu  se necesitan Policurvas para encontrar el perimetro de todo el nivel
       por ello falla cuando existen huecos, hay que encontrar la manera de usar
       curvas simples y que haga un testeo de los extremos.

    x_max: Variable que guarda el l mite m ximo en direcci n X de los puntos de curvas
      perimetrales por cada nivel para realizar comparaci n entre nodos de reticula.
    y_max: Variable que guarda el l mite m ximo en direcci n Y de los puntos de curvas
      perimetrales por cada nivel para realizar comparaci n entre nodos de reticula.
    spa_x: spaCING_x Variable utilizada para obtener el espaciamiento en direcci n X
    spa_y: spaCING_y Variable utilizada para obtener el espaciamiento en direcci n Y
      *Debido a que las dimensiones utilizadas para la ret cula total es un cuarto de
       de tama o de la real se utilizaron las variables anteriores.
    int_nodes: intERIOR_nodes Vector por cada piso que guarda los nodos que se utili-
      zar n en las columnas interiores, las cuales est n dentro de la curva perime-
      tral.
    aux_point: auxILIAR_point utilizado para guardar un punto de la ret cula total,
      el cual mediante un ciclo ser comparado con los l mites definidos en la
      curva perimetral.
    grid_nodes: Vector por cada nivel que guarda la ret cula completa, la cual ser  ut-
      ilizado para obtener las vigas interiores.
      *Checar si se puede omitir los ciclos y asignar esta variable directa del
      arreglo de grid!!!
    int_nodes_trim: intERIOR_nodes_trim vector por nivel que resulta del recorte o Trim
      entre los puntos de la ret cula y las superficies de cada piso.
    int_nodes_flatrim: intERIOR_nodes_flatTENtrim utilizado para eliminar vectores vac os
      debidos a la funci n de trim.
    aux_list: auxILIAR_list vector por nivel que ayuda a ingresar los nodos por nivel
      a la variable int_nodes_floor.
  """

  ##Tiempo ini
  local1_time_ini=time.time()

  n_floors=final_f-first_f
  #Separaci n en entrepisos
  spa=height/n_floors

  #Asignaci n de un tama o de reticula inicial fijo (Una planta equivale a las distancias
  #de un cuarto de planta real)
  x_max_grid=ret[0] #Tama o total de ret
  y_max_grid=ret[1]

  #Arreglo matricial de nodos por cada nivel
  t_beams_floor=[]
  t_nodes_floor=[]

  #Copia de nodos para no tener que partir los for de floors
  grid_nodes_floor_x=[]
  grid_nodes_floor_y=[]

  peri_beams=[] #Arreglo auxiliar que guardar  por cada piso las vigas externas


  ###Esto apenas es para obtener los nodos que se usaran para generar las vigas!!!!
  #final_f m s 1 para considerar la azotea
  for f in range(first_f,final_f+1):

    grid_nodes_floor_x.append(sorted(grid_nodes_floor[f], key=lambda x: x.X))
    grid_nodes_floor_y.append(sorted(grid_nodes_floor[f], key=lambda x: x.Y))


    t_beams=[] #Arreglo auxiliar que guardar  por cada piso las vigas, para posteriormente guardarlo en una matriz

    #Vigas con direcci n X
    for i in range(grid_nodes_floor_x[f].Count-1):

      if round(grid_nodes_floor_x[f][i].X,8) == round(grid_nodes_floor_x[f][i+1].X,8):
        t_beams.append(Line.ByStartPointEndPoint(grid_nodes_floor_x[f][i], grid_nodes_floor_x[f][i+1]))

    #Vigas con direcci n Y
    for i in range(grid_nodes_floor_y[f].Count-1):

      if round(grid_nodes_floor_y[f][i].Y,8) == round(grid_nodes_floor_y[f][i+1].Y,8):
        t_beams.append(Line.ByStartPointEndPoint(grid_nodes_floor_y[f][i], grid_nodes_floor_y[f][i+1]))



    ### Obtenci n de nodos de acuerdo a las intersecciones de vigas (reticula mayor a planta)
    ### Esta funcionando solo con las lineas que tenian problema (por ahora eso esta bien)
    inter_nodes=[]
    inter_nodes_flat=[]

    for i in range (t_beams.Count):
```

```
593
594         if curves[f].DoesIntersect(
            t_beams[i]):
595
596             inter_nodes.append(curves[f].
            Intersect(t_beams[i]))
597
598
599         for i in range (inter_nodes.Count):
600           for j in range (inter_nodes[i].
            Count):
601             inter_nodes_flat.append(
            inter_nodes[i][j])
602
603
604
605         #Filtrado de puntos para obtener
            solo lineas
606         inter_nodes_flat_filt=[]
607         #nodes_prueba_filt=[]
608         inter_nodes_flat_filt=list_filter(
            inter_nodes_flat,"Point")
609
610
611         #Se debe verificar que no se
            repitan nodos
612         #Caso cuando la reticula toca solo
            unas lineas perimetrales de la
            planta y otras no
613         for i in range(
            inter_nodes_flat_filt.Count):
614           peri_nodes_flag=0
615           for j in range(perimetral_points[
            f].Count):
616             if inter_nodes_flat_filt[i].
            IsAlmostEqualTo(perimetral_points[f
            ][j]):
617               peri_nodes_flag=1
618
619           if peri_nodes_flag==0:
620             perimetral_points[f].append(
            inter_nodes_flat_filt[i])
621
622
623         peri_beams.append(ext_beams(curves[
            f],perimetral_points[f]))
624
625         gen_nodes=[]
626
627         for i in perimetral_points[f]:
628           gen_nodes.append(i)
629
630
631         for i in range(grid_nodes_floor[f].
            Count):
632           gen_nodes_flag=0
633           for j in range(gen_nodes.Count):
634             if grid_nodes_floor[f][i].
            IsAlmostEqualTo(gen_nodes[j]):
635               gen_nodes_flag=1
636
637           if gen_nodes_flag==0:
638             gen_nodes.append(
            grid_nodes_floor[f][i])
639
640
641         #identificaci n de nodos que esten
            dentro de la superficie
642         t_nodes_trim=[]
643         for i in gen_nodes:
644           t_nodes_trim.append(i.Intersect(
            surfaces[f]))
645
646
647         #Flatten para eliminar emptylist
648         t_nodes_flatrim=[]
649         for i in range(t_nodes_trim.Count):
650           for j in range(t_nodes_trim[i].
            Count):
651             t_nodes_flatrim.append(
            t_nodes_trim[i][j])
652
653         aux_list=[]
654
655         for i in range(t_nodes_flatrim.
            Count):
656           aux_list.append(t_nodes_flatrim[i
            ])
657
658
659
660         #Filtrado de puntos para obtener
            solo puntos
661         aux_filt=[]
662         aux_filt=list_filter(aux_list,"
            Point")
663
664         ### Nodos totales por piso
665         t_nodes_floor.append(aux_filt)
666
667       ##Tiempo fin
668       local1_time_fin.append(time.time()  -
            local1_time_ini)
669
670
671
672       ##Tiempo ini
673       local2_time_ini=time.time()
674
675       t_nodes_floor_x=[]
676       t_nodes_floor_y=[]
677
678       ### Obtenci n de vigas a partir de
            nodos anteriores
679
680       #Se utilizaron las funciones bubble
            porque adem s de acomodar en X
            tambi n las ordena
681       # en sub listas por Y (Y viceversa
            para el ordenamiento en Y)
682       #t_nodes_floor_x=x_align_nodes(
            t_nodes_floor,first_f,final_f)
683       #t_nodes_floor_y=y_align_nodes(
            t_nodes_floor,first_f,final_f)
684
685       ## Ordenaci n de nodos en X,
            considerando sublistas ordenadas en
            Y
686       for f in range(n_floors+1):
687         t_nodes_floor_x_pre=[]
688         t_nodes_floor_x_pre=sorted(
            t_nodes_floor[f], key=lambda x: x.X
            )
689
690         aux_x=[]
691         sort_chop_p=t_nodes_floor_x_pre[0]
692         sort_chop=[]
693         sort_chop_pre=[]
694         for i in range(t_nodes_floor_x_pre.
            Count):
695
696           if round(sort_chop_p.X, 4)==round
            (t_nodes_floor_x_pre[i].X,4):
697             sort_chop_pre.append(
            t_nodes_floor_x_pre[i])
698
699           else:
700             sort_chop_pre.sort(key=lambda x
            : x.Y)
701             sort_chop.append(sort_chop_pre)
702             sort_chop_pre=[]
703             sort_chop_p=t_nodes_floor_x_pre
            [i]
704
705             sort_chop_pre.append(
            t_nodes_floor_x_pre[i])
706
707         #Al final hay que pasar los ultimos
            nodos
708         sort_chop_pre.sort(key=lambda x: x.
            Y)
709         sort_chop.append(sort_chop_pre)
710
711         for i in sort_chop:
712           for j in i:
713             aux_x.append(j)
714
715         t_nodes_floor_x.append(aux_x)
```

```
716
717   ## Ordenaci n de nodos en Y,
          considerando sublistas ordenadas en
          X
718   for f in range(n_floors+1):
719     t_nodes_floor_y_pre=[]
720     t_nodes_floor_y_pre=sorted(
          t_nodes_floor[f], key=lambda x: x.Y
          )
721
722     aux_x=[]
723     sort_chop_p=t_nodes_floor_y_pre[0]
724     sort_chop=[]
725     sort_chop_pre=[]
726     for i in range(t_nodes_floor_y_pre.
          Count):
727
728       if round(sort_chop_p.Y, 4)==round
          (t_nodes_floor_y_pre[i].Y,4):
729         sort_chop_pre.append(
          t_nodes_floor_y_pre[i])
730
731       else:
732         sort_chop_pre.sort(key=lambda x
          : x.X)
733         sort_chop.append(sort_chop_pre)
734         sort_chop_pre=[]
735         sort_chop_p=t_nodes_floor_y_pre
          [i]
736
737         sort_chop_pre.append(
          t_nodes_floor_y_pre[i])
738
739     #Al final hay que pasar los ultimos
          nodos
740     sort_chop_pre.sort(key=lambda x: x.
          X)
741     sort_chop.append(sort_chop_pre)
742
743     for i in sort_chop:
744       for j in i:
745         aux_x.append(j)
746
747     t_nodes_floor_y.append(aux_x)
748
749   ##Criterio de viga m s larga
750   # Se limitar  a la diagonal de la
          reticula (evita comportamientos
          raros en sup curvas)
751   side_1=2*x_max_grid/spa_in[0]
752   side_2=2*y_max_grid/spa_in[1]
753
754   longest_beam=sqrt(side_1*side_1+
          side_2*side_2)
755
756
757   #final_f m s 1 para considerar la
          azotea
758   for f in range(first_f,final_f+1):
759
760
761     t_beams=[] #Arreglo auxiliar que
          guardar  por cada piso las vigas,
          para posteriormente guardarlo en
          una matriz
762     #Vigas con direcci n X
763     for i in range(t_nodes_floor_x[f].
          Count-1):
764
765       if round(t_nodes_floor_x[f][i].X
          ,3) == round(t_nodes_floor_x[f][i
          +1].X,3):
766         l_aux=Line.ByStartPointEndPoint
          (t_nodes_floor_x[f][i],
          t_nodes_floor_x[f][i+1])
767         if l_aux.Length <= longest_beam
          :
768           t_beams.append(l_aux)
769
770     #Vigas con direcci n Y
771     for i in range(t_nodes_floor_y[f].
          Count-1):
772
773       if round(t_nodes_floor_y[f][i].Y
          ,3) == round(t_nodes_floor_y[f][i
          +1].Y,3):
774         l_aux=Line.ByStartPointEndPoint
          (t_nodes_floor_y[f][i],
          t_nodes_floor_y[f][i+1])
775         if l_aux.Length <= longest_beam
          :
776           t_beams.append(l_aux)
777
778     t_beams_floor.append(t_beams)
779
780     for i in range(peri_beams[f].Count)
          :
781       ext_beams_flag=0
782       for j in range(t_beams_floor[f].
          Count):
783         if peri_beams[f][i].
          IsAlmostEqualTo(t_beams_floor[f][j
          ]):
784           ext_beams_flag=1
785
786       if ext_beams_flag==0:
787         t_beams_floor[f].append(
          peri_beams[f][i])
788
789
790
791   ##Tiempo fin
792   local2_time_fin.append(time.time()  -
          local2_time_ini)
793
794
795
796   ##Tiempo ini
797   local3_time_ini=time.time()
798
799
800   index_1=0
801   index_2=0
802
803
804   #final_f m s 1 para considerar la
          azotea
805   for f in range(first_f,final_f+1):
806
807     #Agregado de objetos frame
808     aux_fobj_list=[]
809     #Lista de indices
810     aux_find_obj_list=[]
811
812     for i in range(t_beams_floor[f-
          first_f].Count):
813       aux_fobj_list.append(frame(
          index_1))
814       aux_fobj_list[i].add_frame(
          t_beams_floor[f-first_f][i])
815       #Obtenci n de  ndices  para
          frames
816       aux_find_obj_list.append(
          aux_fobj_list[i].id)
817
818       index_1=index_1+1
819
820     #Lista de objetos bframe
821     bframes_obj_vec.append(
          aux_fobj_list)
822     #Lista de indices de objeto bframe
823     bframes_ind_vec.append(
          aux_find_obj_list)
824
825
826   t_nodes_floor_forobj=[]
827
828   #final_f m s 1 para considerar la
          azotea
829   for f in range(first_f,final_f+1):
830
831     t_nodes_floor_forobj.append(
          t_nodes_floor_x[f])#x_align_nodes(
          t_nodes_floor,first_f,final_f)
832
833     #Agregado de objetos nodo
834     aux_nobj_list=[]
```

```
    #Lista de indices
    aux_nind_obj_list=[]

    #Asignaci n de nodos (vector simple) a vector de objetos nodo
    for i in range(t_nodes_floor_forobj[f-first_f].Count):
      aux_nobj_list.append(node(index_2))
      aux_nobj_list[i].add_point(t_nodes_floor_forobj[f-first_f][i])
      #Obtenci n de  ndices  para nodos
      aux_nind_obj_list.append(aux_nobj_list[i].id)

      index_2=index_2+1

    #Lista de objetos nodo
    nodes_obj_vec.append(aux_nobj_list)

    #Lista de indices de objeto nodo
    nodes_ind_vec.append(aux_nind_obj_list)



  #Llama a funci n de conectividades
  node_beam_index=search_nodes_frames(bframes_ind_vec,bframes_obj_vec,nodes_ind_vec,nodes_obj_vec)

  ##Tiempo fin
  local3_time_fin.append(time.time() - local3_time_ini)

  #Reacomodo de indices en barras para eliminaci n de primer nivel (planta baja sin barras)
  n_index_floor_0=len(bframes_ind_vec[0])
  for f in range(1,final_f+1):
    for i in range(bframes_ind_vec[f].Count ):
      bframes_ind_vec[f][i]=bframes_ind_vec[f][i]-n_index_floor_0

  #Eliminaci n del primer nivel o planta baja (no tiene vigas)
  bframes_ind_vec.pop(0)
  t_beams_floor.pop(0)
  node_beam_index.pop(0)

  #Libera peri_beams
  del peri_beams

  #Libera t_beams
  del t_beams

  return nodes_ind_vec,t_nodes_floor_x,bframes_ind_vec, t_beams_floor,node_beam_index


def ext_beams(curve,ext_nodes):

  points_para=[]
  for i in ext_nodes:
    points_para.append(curve.ParameterAtPoint(i))

  points_para.sort()

  per_sort_points=[]

  for i in range(points_para.Count):
    per_sort_points.append(curve.PointAtParameter(points_para[i]))

  per_beams=[]
  for i in range(per_sort_points.Count-1):
    per_beams.append(Line.ByStartPointEndPoint(per_sort_points[i], per_sort_points[i+1]))

  per_beams.append(Line.ByStartPointEndPoint(per_sort_points[per_sort_points.Count-1], per_sort_points[0]))

  return per_beams

def list_filter(list, filter):
  """
  Descripci n:
    Funci n auxiliar que ayuda a limpiar listas que contengan elementos no deseados,
    como por ejemplo cuando se crean intersecciones entre geometr as y se desea ob-
    tener puntos, muchas veces se crean otros elementos como l neas o "emptylist"
    que nos dificultan para un proceso subsiguiente, con esta funci n se eliminan
    y nos devuelve una lista con el mismo tipo de elemento.

  Argumentos:
    list: Lista a limpiar
    filter: Cadena de texto con el nombre del elemento a conservar en la lista usando
      una funci n denominada GetType().

  Variables:
    list_filtered: Lista filtrada final.

  """
  list_filtered=[]
  for l in list:
    if filter in str(l.GetType()):
      list_filtered.append(l)
  return list_filtered

def x_align_nodes(nodes, first_f, final_f):
  ordered_vec_f=[]

  #bubble sort con .X
  for f in range(first_f,final_f+1):
    for i in range(nodes[f-first_f].Count):
      for j in range(i,nodes[f-first_f].Count):

        if nodes[f-first_f][i].X>nodes[f-first_f][j].X and round(nodes[f-first_f][i].Z,8) == round(nodes[f-first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].X,8)==round(nodes[f-first_f][j].X,8):
          counter=counter+1
```

```
            aux_order_vec.append(nodes[f-first_f][j])
            j=j+1

        for k in range(aux_order_vec.Count):
            for l in range(k,aux_order_vec.Count):
                if aux_order_vec[k].Y>aux_order_vec[l].Y:
                    aux= aux_order_vec[l]
                    aux_order_vec[l]=aux_order_vec[k]
                    aux_order_vec[k]=aux

        ordered_vec.append(aux_order_vec)
        i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
        for j in range(ordered_vec[i].Count):
            ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f

def y_align_nodes(nodes, first_f, final_f):
  ordered_vec_f=[]

  #bubble sort con .Y
  for f in range(first_f,final_f+1):
    for i in range(nodes[f-first_f].Count):
      for j in range(i,nodes[f-first_f].Count):

        if nodes[f-first_f][i].Y>nodes[f-first_f][j].Y and round(nodes[f-first_f][i].Z,8) == round(nodes[f-first_f][j].Z,8):
          aux= nodes[f-first_f][j]
          nodes[f-first_f][j]= nodes[f-first_f][i]
          nodes[f-first_f][i]=aux

    ordered_vec=[]
    counter=0
    i=0
    j=0
    while(i<nodes[f-first_f].Count):
      node_i=nodes[f-first_f][i]
      counter=0
      aux_order_vec=[]
      j=i
      while(j<nodes[f-first_f].Count):
        if round(nodes[f-first_f][i].Y,8)==round(nodes[f-first_f][j].Y,8):
          counter=counter+1
          aux_order_vec.append(nodes[f-first_f][j])
        j=j+1

      for k in range(aux_order_vec.Count):
        for l in range(k,aux_order_vec.Count):
          if aux_order_vec[k].X>aux_order_vec[l].X:
            aux= aux_order_vec[l]
            aux_order_vec[l]=aux_order_vec[k]
            aux_order_vec[k]=aux

      ordered_vec.append(aux_order_vec)
      i=i+counter

    ordered_vec_flat=[]

    for i in range(ordered_vec.Count):
      for j in range(ordered_vec[i].Count):
        ordered_vec_flat.append(ordered_vec[i][j])

    ordered_vec_f.append(ordered_vec_flat)


  return ordered_vec_f

def search_nodes_frames(index_frames_vec,frames_vec,index_nodes_vec,nodes_vec):


  for f in range(frames_vec.Count):

    nodes_vec_flat=[]

    for i in range(nodes_vec[f].Count):
      nodes_vec_flat.append(nodes_vec[f][i])

    if f < frames_vec.Count-1:
      for i in range(nodes_vec[f+1].Count):
        nodes_vec_flat.append(nodes_vec[f+1][i])

    for i in range(frames_vec[f].Count):
      j=0
      ini_flag=0
      fin_flag=0
      nodes_counter=nodes_vec_flat.Count
      sol_flag=0
      while(j<nodes_counter and sol_flag==0):

        if frames_vec[f][i].frame.StartPoint.IsAlmostEqualTo(nodes_vec_flat[j].point):
          ini_flag=1
          frames_vec[f][i].add_ini_point(nodes_vec_flat[j].id)

        elif frames_vec[f][i].frame.EndPoint.IsAlmostEqualTo(nodes_vec_flat[j].point):
          fin_flag=1
          frames_vec[f][i].add_fin_point(nodes_vec_flat[j].id)

        sol_flag=ini_flag*fin_flag
        j=j+1

  conectivities=[]

  for f in range(frames_vec.Count):
    conec_aux=[]
    for s in range(frames_vec[f].Count):
      conec_aux.append([frames_vec[f][s].ini_point,frames_vec[f][s].fin_point])

    conectivities.append(conec_aux)

  return conectivities

def material(beams_mat, columns_mat, beam_index, column_index, first_f, final_f):

  beams_mat_vec=[]
  columns_mat_vec=[]


  mat_analysis_inf=[]
```

```
1076
1077    ##Informacion de cada material, usado en la entrada de datos
1078    #VIGAS
1079    for i in range(beams_mat.Count):
1080        mat_analysis_inf.append(beams_mat[i][3])
1081    #COLUMNAS
1082    for i in range(columns_mat.Count):
1083        mat_analysis_inf.append(columns_mat[i][3])
1084
1085    ##Informacion del material para cada barra
1086    n_floors=final_f-first_f
1087    for f in range(n_floors):
1088        beams_mat_vec_aux=[]
1089        columns_mat_vec_aux=[]
1090
1091        #VIGAS
1092        n_beams_f=beam_index[f].Count
1093        for i in range(n_beams_f):
1094            mat_flag=0
1095            m=0
1096            while(mat_flag==0 and m<beams_mat.Count):
1097                if f >= beams_mat[m][1]-1 and f <= beams_mat[m][2]-1:
1098                    beams_mat_vec_aux.append([m+1,beams_mat[m][0]])
1099                    mat_flag=1
1100
1101                m=m+1
1102
1103        #COLUMNAS
1104        n_cols_f=column_index[f].Count
1105        for i in range(n_cols_f):
1106            mat_flag=0
1107            m=beams_mat.Count
1108            while(mat_flag==0 and m<mat_analysis_inf.Count):
1109                if f >= columns_mat[m-beams_mat.Count][1]-1 and f <= columns_mat[m-beams_mat.Count][2]-1:
1110                    columns_mat_vec_aux.append([m+1,columns_mat[m-beams_mat.Count][0]])
1111                    mat_flag=1
1112
1113                m=m+1
1114
1115
1116        beams_mat_vec.append(beams_mat_vec_aux)
1117        columns_mat_vec.append(columns_mat_vec_aux)
1118
1119    return mat_analysis_inf, beams_mat_vec, columns_mat_vec
```

A.7. Código: Distribución de Cargas

```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import math
import itertools
import time

#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

#Variables globales
first_f=0
final_f=IN[0][0][0][0][2]

beams_inf=IN[0][1]

"""///*** Main ***///"""
#Vector con informacion de todos los pisos
#cargas_todos_pisos=[]

#Vector de superficies usadas para color
color_surfaces=[]
#Vector de indices de color de acuerdo al area
color_index=[]
beam_loads=[]

#Carga de dise o
loads=IN[1]

#Informacion de la reticula
ret_data=IN[0][0][0]

centroids=ret_data[2]

#Cargas en kg/m2, por eso la divisi n
load_factor=0
for i in loads:
  load_factor=load_factor+i/10000.0

global_time_ini=time.time()

for f in range(first_f,final_f):
  global_time_1_ini=time.time()
  #inicializacion vector de objetos
  slabs_vec=[]

  #Informacion necesaria que cambia con los pisos
  beam_points=beams_inf[1][1][f+1]
  beams=beams_inf[1][3][f]
  beams_index=beams_inf[1][2][f]

  #sep_slabs=[]
  #cargas_por_piso=[]
  ordered_beams=[]

  #Superficies de losas a partir de la funcion get_slab_ret en ret
  ret_slabs=ret_data[3][f+1]

  fun1_time_ini=time.time()
  tributary_areas=get_tribu_areas(ret_slabs,centroids[f+1], beam_points)
  fun1_time_fin=time.time() - fun1_time_ini

  fun2_time_ini=time.time()
  ordered_beams=search_count_beams(beams, beams_index)
  fun2_time_fin=time.time() - fun2_time_ini
  ##Informaci n no tan importante para calculo
  #Descomentar si es necesario observar esta inf
  #cargas_por_piso.append(sep_slabs)
  #cargas_por_piso.append(tributary_areas)
  #cargas_por_piso.append(ordered_beams)
  #cargas_todos_pisos.append(cargas_por_piso)

  ## Asignaci n de cargas
  beam_loads_aux=[]
  slab_floor_vec=[]
  slab_floor_vec=slabs_vec


  fun3_time_ini=time.time()
  beam_loads_aux=load_assign(beams, beams_index, slab_floor_vec)
  fun3_time_fin=time.time() - fun3_time_ini

  beam_loads.append(beam_loads_aux)

  ## Rango de Colores
  color_surfaces.append(tributary_areas[2])
  color_index.append(tributary_areas[3])

  global_time_1_fin=time.time() - global_time_1_ini


color_func=[color_surfaces,color_index]
global_time_fin=time.time() - global_time_ini
times=[global_time_fin, global_time_1_fin,fun1_time_fin, fun2_time_fin,fun3_time_fin]
conec_parameters=[beams_inf]
#Assign your output to the OUT variable.
OUT = conec_parameters,color_func, beam_loads, ordered_beams, times

"""///*** Functions ***///"""

# Objeto Losa - - - - - - - - - - - - - - - - - - -
# __init__      --
# Construye objeto y asigna espacio para listas
# add_point     --
# Agrega un punto (4) que define el contorno de
# la losa
# add_tri_points --
# Agrega un punto (2) que define las areas de
# triagulos  y trapecios                    #
# add_tri_curve  --
# Agrega cada curva que forman los tri ngulos y
# trapecios
# add_tri_area   --
# Obtiene cada area (magnitud) de los areas
# tributarias
# add_cont_beams --
# Agrega indice de cada viga exterior a la losa
# - - - - - - - - - - - - - - - - - - - - - - - - -



class slab:
```

```
123
124   def __init__(self, id):
125     self.id=id
126     self.surf_points_vec=[]
127     self.tri_points_vec=[]
128     self.tri_curves_vec=[]
129     self.tri_surf_vec=[]
130     self.tri_areas_vec=[]
131     self.tri_areas_color_vec=[]
132     self.c_beams_vec=[]
133     self.trisubsurf_vec=[]
134
135
136   def add_point(self, surf_point):
137     self.surf_points_vec.append(
        surf_point)
138
139   def add_tri_points(self, tri_point):
140     self.tri_points_vec.append(
        tri_point)
141
142   def add_tri_curve(self,tri_curve):
143     self.tri_curves_vec.append(
        tri_curve)
144
145
146   def add_tri_area(self,tri_curve):
147
148     tri_surf_points=[]
149     #Al agregar uno primero, se evita
        que haya coincidencia
150     #de puntos, se debe verificar
        adentro del ciclo
151
152     for i in range(tri_curve.Count):
153
154       tri_surf_points.append(tri_curve[
        i].StartPoint)
155
156     tri_surf=Surface.ByPerimeterPoints(
        tri_surf_points)
157
158     #Area interior de las vigas
159     #Se multiplica por 10 000 para
        tenerlo en cm2
160     tri_area=tri_surf.Area*10000
161
162
163
164     self.tri_surf_vec.append(tri_surf)
165     self.tri_areas_vec.append(tri_area)
166
167   def add_tri_area_color(self,tri_color
        ):
168     self.tri_areas_color_vec.append(
        tri_color)
169
170
171   def add_cont_beams(self, c_beam):
172     self.c_beams_vec.append(c_beam)
173
174   def add_centroid(self, c_point):
175     self.centroid=c_point
176
177   def add_subsurf(self, subsurf):
178     self.subsurf=subsurf
179
180   def add_trisubsurf(self, trisubsurf):
181     self.subsurf_vec.append(trisubsurf)
182
183   def __del__(self):
184     print #Destructor
185
186 def get_tribu_areas(surf_vec,centroids,
        beam_points):
187
188
189   for i in range(surf_vec.Count):
190     peri_beams_aux=[]
191
192     slabs_vec.append(slab(i))
193     #Se guardaran las superficies
194     #Por si se ocupan despu s
195     #quiz  no se ocupen porque ya los
        tenemos
196     slabs_vec[i].add_subsurf(surf_vec[i
        ])
197
198
199     #Busqueda de barras perimetrales *
        optimizable
200     for j in range(beams.Count):
201       if beams[j].StartPoint.
        DoesIntersect(surf_vec[i]) and
        beams[j].EndPoint.DoesIntersect(
        surf_vec[i]):
202         peri_beams_aux.append(beams[j])
203
204     dir_flag=0
205     #Verificaci n de p ligonos
        peri_beams_aux!= 4
206     if peri_beams_aux.Count==4:
207
208       #Asignaci n de nodos extremos,
        los puntos son acomodados
        previamente para que funcione segun
        ref.
209
210       vert_nodes=[]
211       vert_nodes_aligned=[]
212       vert_aux=[]
213       curves_rec_aux=[]
214       #Nodo 0
215
216       for k in range(peri_beams_aux.
        Count):
217         vert_snode_flag=0
218         vert_enode_flag=0
219         for l in range(vert_nodes.Count
        ):
220
221           if peri_beams_aux[k].
        StartPoint.IsAlmostEqualTo(
        vert_nodes[l]):
222             vert_snode_flag=1
223           if peri_beams_aux[k].EndPoint
        .IsAlmostEqualTo(vert_nodes[l]):
224             vert_enode_flag=1
225
226
227         if vert_snode_flag==0:
228           vert_nodes.append(
        peri_beams_aux[k].StartPoint)
229           vert_aux.append(
        peri_beams_aux[k].StartPoint)
230         if vert_enode_flag==0:
231           vert_nodes.append(
        peri_beams_aux[k].EndPoint)
232           vert_aux.append(
        peri_beams_aux[k].EndPoint)
233
234
235       #Areas tributarias formadas por
        trapecios y triangulos
236       #Se manda llamar la funci n de
        get_tribu_areas_rec_surf
237
238       vert_nodes_aligned=slab_align(
        vert_nodes)
239
240       for n in vert_nodes_aligned:
241         #puntos
242         slabs_vec[i].add_point(n)
243
244       #Verificacion de direccion
245       for d in range(vert_nodes_aligned
        .Count-2):
246         vec_1=Vector.ByTwoPoints(
        vert_nodes_aligned[d],
        vert_nodes_aligned[d+1])
247         vec_2=Vector.ByTwoPoints(
        vert_nodes_aligned[d+1],
        vert_nodes_aligned[d+2])
248         ang_dir=abs(vec_1.AngleBetween(
        vec_2))
249
```

```
        if round(ang_dir,1)==0 or round(ang_dir,1)==0:
            dir_flag=1

      if dir_flag==0:
        curves_rec_aux=get_tribu_areas_rec_surf(vert_nodes_aligned,i)

        #Curvas
        slabs_vec[i].add_tri_curve(curves_rec_aux[0])
        slabs_vec[i].add_tri_curve(curves_rec_aux[1])
        slabs_vec[i].add_tri_curve(curves_rec_aux[2])
        slabs_vec[i].add_tri_curve(curves_rec_aux[3])

        #Areas
        slabs_vec[i].add_tri_area(curves_rec_aux[0])
        slabs_vec[i].add_tri_area(curves_rec_aux[1])
        slabs_vec[i].add_tri_area(curves_rec_aux[2])
        slabs_vec[i].add_tri_area(curves_rec_aux[3])

      elif dir_flag==1:

        #triangulacion por perimetro-centroide
        for j in range(peri_beams_aux.Count):
            tri_lines_aux=[]
            tribu_line_1=Line.ByStartPointEndPoint(peri_beams_aux[j].EndPoint,centroids[i])
            tribu_line_2=Line.ByStartPointEndPoint(centroids[i],peri_beams_aux[j].StartPoint)
            tri_lines_aux=[tribu_line_1,tribu_line_2,peri_beams_aux[j]]


            #Curvas
            slabs_vec[i].add_tri_curve(tri_lines_aux)
            #Areas
            slabs_vec[i].add_tri_area(tri_lines_aux)

    else:

      #puntos perimetrales
      aux_points=[]

      for j in range(peri_beams_aux.Count):
        aux_points.append(peri_beams_aux[j].StartPoint)
        aux_points.append(peri_beams_aux[j].EndPoint)

      j=0
      k=1
      aux_points_counter=aux_points.Count
      while(j<aux_points_counter):
        aux_p=aux_points[j]
        k=j+1
        while(k<aux_points_counter):
          if round(aux_p.X,5)==round(aux_points[k].X,5) and round(aux_p.Y,5)==round(aux_points[k].Y,5):
            #if aux_p.IsAlmostEqualTo(aux_points[k]):
            aux_points.pop(k)
            aux_points_counter=aux_points.Count

          k=k+1
        j=j+1


      for j in aux_points:
        #puntos
        slabs_vec[i].add_point(j)


      #triangulacion por perimetro-centroide
      for j in range(peri_beams_aux.Count):
        tri_lines_aux=[]
        tribu_line_1=Line.ByStartPointEndPoint(peri_beams_aux[j].EndPoint,centroids[i])
        tribu_line_2=Line.ByStartPointEndPoint(centroids[i],peri_beams_aux[j].StartPoint)
        tri_lines_aux=[tribu_line_1,tribu_line_2,peri_beams_aux[j]]


        #Curvas
        slabs_vec[i].add_tri_curve(tri_lines_aux)
        #Areas
        slabs_vec[i].add_tri_area(tri_lines_aux)



## Asignaci n a vectores para output
tri_curves_aux=[]
# Vector de lineas que definen areas de distribucion
#Obtenida de los objetos slab - OPTIMIZABLE
for i in range(slabs_vec.Count):
  tri_curves_aux.append(slabs_vec[i].tri_curves_vec)

tri_surf_aux=[]
# Vector de superficies que definen areas de distribucion
#Obtenida de los objetos slab - OPTIMIZABLE
for i in range(slabs_vec.Count):
  tri_surf_aux.append(slabs_vec[i].tri_surf_vec)

tri_area_aux=[]
# Vector de superficies con los valores de area
#Obtenida de los objetos slab - OPTIMIZABLE
#Esto se hace temporalmente para meter en un vector las areas y puntos de la clase slabs
for s in range(slabs_vec.Count):
  tri_area_aux.append(slabs_vec[s].tri_areas_vec)


#Este es el rango de colores basado en la magnitud de  rea , hacerlo por el peso
min_color=0#slabs_vec[0].tri_areas_vec[0]
max_color=0#slabs_vec[0].tri_areas_vec[0]

#Identificacion de maximo y minimo valor de area para realizar la interpolacion lineal de color
for c in range(slabs_vec.Count):
  for o in range(slabs_vec[c].tri_areas_vec.Count):
    if slabs_vec[c].tri_areas_vec[o] > max_color:
      max_color=slabs_vec[c].tri_areas_vec[o]

    if slabs_vec[c].tri_areas_vec[o] < min_color:
```

```
            min_color=slabs_vec[c].
        tri_areas_vec[o]


    #Interpolacion de color
    for c in range(slabs_vec.Count):
      for o in range(slabs_vec[c].
        tri_areas_vec.Count):
        slabs_vec[c].add_tri_area_color((
        slabs_vec[c].tri_areas_vec[o]-
        max_color)/(max_color-min_color)
        +1.0)

    color_values_aux=[]
    # Vector de valores de color
    #Obtenida de los objetos slab -
       OPTIMIZABLE
    for s in range(slabs_vec.Count):
      for r in range(slabs_vec[s].
        tri_areas_color_vec.Count):
        color_values_aux.append(slabs_vec
        [s].tri_areas_color_vec[r])



    surf_point_aux=[]
    for i in range(slabs_vec.Count):
      surf_point_aux.append(slabs_vec[i].
        surf_points_vec)


    #Libera peri_beams_aux
    #for i in peri_beams_aux:
    # i.Dispose()

    #Libera curves_rec_aux
    del curves_rec_aux

    return  tri_curves_aux, tri_area_aux,
        tri_surf_aux, color_values_aux,
        surf_point_aux

def get_tribu_areas_rec_surf(vert_nodes
    ,i):


    vec_x=Vector.ByCoordinates(1,0,0)
    vec_y=Vector.ByCoordinates(0,1,0)
    vec_01=Vector.ByTwoPoints(vert_nodes
       [0],vert_nodes[1])
    vec_02=Vector.ByTwoPoints(vert_nodes
       [0],vert_nodes[2])
    vec_23=Vector.ByTwoPoints(vert_nodes
       [2],vert_nodes[3])
    vec_13=Vector.ByTwoPoints(vert_nodes
       [1],vert_nodes[3])

    vec_10=Vector.Reverse(vec_01)
    vec_20=Vector.Reverse(vec_02)
    vec_32=Vector.Reverse(vec_23)
    vec_31=Vector.Reverse(vec_13)

    length_02=vec_02.Length
    length_13=vec_13.Length

    length_01=vec_01.Length
    length_23=vec_23.Length


    length_20=vec_20.Length
    length_31=vec_31.Length

    length_10=vec_10.Length
    length_32=vec_32.Length


    """
    Caso 1: Lado m s corto (o igual)
       casi paralelo al eje X

    Caso 2: Lado m s corto casi paralelo
        al eje Y
    """

  if (length_02*length_13<length_01*
     length_23):

      LY02=abs(vert_nodes[0].Y-vert_nodes
       [2].Y)
      LX02=abs(vert_nodes[0].X-vert_nodes
       [2].X)
      LY13=abs(vert_nodes[1].Y-vert_nodes
       [3].Y)
      LX13=abs(vert_nodes[1].X-vert_nodes
       [3].X)


      def ang_compi_x():
        alfa_ang_i=vec_01.AngleBetween(
       vec_02)*math.pi/180
        beta_ang_i=vec_20.AngleBetween(
       vec_23)*math.pi/180
        gamma_i_1=abs(vec_y.AngleBetween(
       vec_01)*math.pi/180)
        gamma_i_2=abs(vec_y.AngleBetween(
       vec_23)*math.pi/180)


        if vert_nodes[1].X<vert_nodes[0].
       X and round(vert_nodes[3].X,1)==
       round(vert_nodes[2].X,1):
          theta_1_i=alfa_ang_i*0.5+
       gamma_i_1
          theta_2_i=beta_ang_i*0.5

        elif round(vert_nodes[3].X,1)==
       round(vert_nodes[2].X,1):
          theta_1_i=alfa_ang_i*0.5#+
       gamma_i_1
          theta_2_i=beta_ang_i*0.5

        if vert_nodes[3].X<vert_nodes[2].
       X and round(vert_nodes[0].X,1)==
       round(vert_nodes[1].X,1):
          theta_1_i=alfa_ang_i*0.5
          theta_2_i=beta_ang_i*0.5#+
       gamma_i_2

        elif round(vert_nodes[1].X,1)==
       round(vert_nodes[0].X,1):
          theta_1_i=alfa_ang_i*0.5
          theta_2_i=beta_ang_i*0.5+
       gamma_i_2

        return theta_1_i, theta_2_i

      def ang_comps_x():
        alfa_ang_s=vec_10.AngleBetween(
       vec_13)*math.pi/180
        beta_ang_s=vec_31.AngleBetween(
       vec_32)*math.pi/180
        gamma_s_1=abs(vec_y.AngleBetween(
       vec_01)*math.pi/180)
        gamma_s_2=abs(vec_y.AngleBetween(
       vec_23)*math.pi/180)

        if vert_nodes[1].X<vert_nodes[0].
       X and round(vert_nodes[3].X,1)==
       round(vert_nodes[2].X,1):
          theta_1_s=alfa_ang_s*0.5#+
       gamma_s_1
          theta_2_s=beta_ang_s*0.5

        elif round(vert_nodes[3].X,1)==
       round(vert_nodes[2].X,1):
          theta_1_s=alfa_ang_s*0.5+
       gamma_s_1
          theta_2_s=beta_ang_s*0.5

        if vert_nodes[3].X<vert_nodes[2].
       X and round(vert_nodes[0].X,1)==
       round(vert_nodes[1].X,1):
          theta_1_s=alfa_ang_s*0.5
          theta_2_s=beta_ang_s*0.5+
       gamma_s_2

        elif round(vert_nodes[1].X,1)==
       round(vert_nodes[0].X,1):
```

```
        theta_1_s=alfa_ang_s*0.5
        theta_2_s=beta_ang_s*0.5#+gamma_s_2

      return theta_1_s, theta_2_s



    #Angulo punto inferior
    if vert_nodes[2].Y<=vert_nodes[0].Y:

      if round(vert_nodes[0].X,1)==round(vert_nodes[1].X,1) and round(vert_nodes[2].X,1)==round(vert_nodes[3].X,1):
        #Radianes
        alfa_ang_i=vec_01.AngleBetween(vec_02)*math.pi/180
        beta_ang_i=vec_20.AngleBetween(vec_23)*math.pi/180
        gamma_i=vec_x.AngleBetween(vec_02)*math.pi/180
        gamma_i_2=vec_y.AngleBetween(vec_01)*math.pi/180-math.pi/180*0.5
        theta_1_i=alfa_ang_i*0.5#+gamma_i_2
        theta_2_i=beta_ang_i*0.5+gamma_i

        angi=[theta_1_i,theta_2_i]

      else:

        angi=ang_compi_x()

      LY02=-LY02

      #Puntos auxiliares para formar los triangulos y los trapecios
      #Se usan expresiones para X y Y global
      aux_point_i=Point.ByCoordinates((LY02+math.tan(angi[1])*LX02)/(math.tan( angi[0]) + math.tan(angi[1]))+ vert_nodes[0].X, \
        math.tan(angi[0])*((LY02+math.tan(angi[1])*LX02)/(math.tan(angi[0])+math.tan(angi[1]))) +  vert_nodes[0].Y, \
          vert_nodes[0].Z)
    else:

      if round(vert_nodes[0].X,1)==round(vert_nodes[1].X,1) and round(vert_nodes[2].X,1)==round(vert_nodes[3].X,1):
        #Radianes
        alfa_ang_i=vec_01.AngleBetween(vec_02)*math.pi/180
        beta_ang_i=vec_20.AngleBetween(vec_23)*math.pi/180
        gamma_i=vec_x.AngleBetween(vec_02)*math.pi/180
        theta_1_i=alfa_ang_i*0.5+gamma_i
        theta_2_i=beta_ang_i*0.5

        angi=[theta_1_i,theta_2_i]

      else:

        angi=ang_compi_x()

      #Puntos auxiliares para formar los triangulos y los trapecios
      #Se usan expresiones para X y Y global
      aux_point_i=Point.ByCoordinates((LY02+math.tan(angi[1])*LX02)/(math.tan( angi[0]) + math.tan(angi[1]))+ vert_nodes[0].X, \
        math.tan(angi[0])*((LY02+math.tan(angi[1])*LX02)/(math.tan(angi[0])+math.tan(angi[1]))) +  vert_nodes[0].Y, \
          vert_nodes[0].Z)
    #Angulo punto superior
    if vert_nodes[3].Y<=vert_nodes[1].Y:

      if round(vert_nodes[0].X,1)==round(vert_nodes[1].X,1) and round(vert_nodes[2].X,1)==round(vert_nodes[3].X,1):

        alfa_ang_s=vec_10.AngleBetween(vec_13)*math.pi/180
        beta_ang_s=vec_31.AngleBetween(vec_32)*math.pi/180
        gamma_s=vec_x.AngleBetween(vec_13)*math.pi/180
        theta_1_s=alfa_ang_s*0.5+gamma_s
        theta_2_s=beta_ang_s*0.5

        angs=[theta_1_s,theta_2_s]

      else:

        angs=ang_comps_x()

      aux_point_s=Point.ByCoordinates((LY13+math.tan(angs[1])*LX13)/(math.tan( angs[0]) + math.tan(angs[1]))+ vert_nodes[1].X, \
        -math.tan(angs[0])*((LY13+math.tan(angs[1])*LX13)/(math.tan(angs[0]) +math.tan(angs[1]))) + vert_nodes[1].Y, \
          vert_nodes[1].Z)

    else:
      if round(vert_nodes[0].X,1)==round(vert_nodes[1].X,1) and round(vert_nodes[2].X,1)==round(vert_nodes[3].X,1):

        alfa_ang_s=vec_10.AngleBetween(vec_13)*math.pi/180
        beta_ang_s=vec_31.AngleBetween(vec_32)*math.pi/180
        gamma_i=vec_x.AngleBetween(vec_13)*math.pi/180
        theta_1_s=alfa_ang_s*0.5
        theta_2_s=beta_ang_s*0.5+gamma_i

        angs=[theta_1_s,theta_2_s]

      else:

        angs=ang_comps_x()

      LY13=-LY13

      aux_point_s=Point.ByCoordinates((LY13+math.tan(angs[1])*LX13)/(math.tan( angs[0]) + math.tan(angs[1]))+ vert_nodes[1].X, \
        -math.tan(angs[0])*((LY13+math.tan(angs[1])*LX13)/(math.tan(angs[0]) +math.tan(angs[1]))) + vert_nodes[1].Y, \
          vert_nodes[1].Z)


    slabs_vec[i].add_tri_points(aux_point_i)
    slabs_vec[i].add_tri_points(aux_point_s)


    #Se obtienen las lineas de cada triangulo y trapecio
    #Triangulo inferior
```

```
    line_aux_11=Line.ByStartPointEndPoint(vert_nodes[0],aux_point_i)
    line_aux_12=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[2])
    line_aux_13=Line.ByStartPointEndPoint(vert_nodes[2],vert_nodes[0])

    #Triangulo superior
    line_aux_21=Line.ByStartPointEndPoint(vert_nodes[1],aux_point_s)
    line_aux_22=Line.ByStartPointEndPoint(aux_point_s,vert_nodes[3])
    line_aux_23=Line.ByStartPointEndPoint(vert_nodes[3],vert_nodes[1])

    #Trapecio
    line_aux_31=Line.ByStartPointEndPoint(vert_nodes[0],vert_nodes[1])
    line_aux_32=Line.ByStartPointEndPoint(vert_nodes[1],aux_point_s)
    line_aux_33=Line.ByStartPointEndPoint(aux_point_s,aux_point_i)
    line_aux_34=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[0])

    #Trapecio
    line_aux_41=Line.ByStartPointEndPoint(vert_nodes[2],vert_nodes[3])
    line_aux_42=Line.ByStartPointEndPoint(vert_nodes[3],aux_point_s)
    line_aux_43=Line.ByStartPointEndPoint(aux_point_s,aux_point_i)
    line_aux_44=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[2])

    curves_aux_3=[line_aux_11,line_aux_12,line_aux_13]
    curves_aux_4=[line_aux_21,line_aux_22,line_aux_23]
    curves_aux_1=[line_aux_31,line_aux_32,line_aux_33,line_aux_34]
    curves_aux_2=[line_aux_41,line_aux_42,line_aux_43,line_aux_44]


elif (length_02*length_13>length_01*length_23):
    #Se cambia posici n de nodo auxiliar para generar tri ngulos y trapecios
    LY01=abs(vert_nodes[0].Y-vert_nodes[1].Y)
    LX01=abs(vert_nodes[0].X-vert_nodes[1].X)
    LY23=abs(vert_nodes[2].Y-vert_nodes[3].Y)
    LX23=abs(vert_nodes[2].X-vert_nodes[3].X)


    def ang_compi_y():
      alfa_ang_i=abs(vec_01.AngleBetween(vec_02)*math.pi/180)
      beta_ang_i=abs(vec_10.AngleBetween(vec_13)*math.pi/180)
      gamma_i_1=abs(vec_x.AngleBetween(vec_02)*math.pi/180)
      gamma_i_2=abs(vec_x.AngleBetween(vec_13)*math.pi/180)

      if vert_nodes[2].Y<vert_nodes[0].Y and round(vert_nodes[3].Y,1)==round(vert_nodes[1].Y,1):
          theta_1_i=alfa_ang_i*0.5#-gamma_i_1
          theta_2_i=beta_ang_i*0.5

      elif round(vert_nodes[3].Y,1)==round(vert_nodes[1].Y,1):
          theta_1_i=alfa_ang_i*0.5+gamma_i_1
          theta_2_i=beta_ang_i*0.5

      if vert_nodes[3].Y<vert_nodes[1].Y and round(vert_nodes[2].Y,1)==round(vert_nodes[0].Y,1):
          theta_1_i=alfa_ang_i*0.5
          theta_2_i=beta_ang_i*0.5+gamma_i_2

      elif round(vert_nodes[2].Y,1)==round(vert_nodes[0].Y,1):
          theta_1_i=alfa_ang_i*0.5
          theta_2_i=beta_ang_i*0.5#-gamma_i_2

      return theta_1_i, theta_2_i

    def ang_comps_y():
      alfa_ang_s=vec_20.AngleBetween(vec_23)*math.pi/180
      beta_ang_s=vec_31.AngleBetween(vec_32)*math.pi/180
      gamma_s_1=abs(vec_x.AngleBetween(vec_02)*math.pi/180)
      gamma_s_2=abs(vec_x.AngleBetween(vec_13)*math.pi/180)

      if vert_nodes[2].Y<vert_nodes[0].Y and round(vert_nodes[3].Y,1)==round(vert_nodes[1].Y,1):
          theta_1_s=alfa_ang_s*0.5+gamma_s_1
          theta_2_s=beta_ang_s*0.5

      elif round(vert_nodes[3].Y,1)==round(vert_nodes[1].Y,1):
          theta_1_s=alfa_ang_s*0.5#-gamma_s_1
          theta_2_s=beta_ang_s*0.5

      if vert_nodes[3].Y<vert_nodes[1].Y and round(vert_nodes[2].Y,1)==round(vert_nodes[0].Y,1):
          theta_1_s=alfa_ang_s*0.5
          theta_2_s=beta_ang_s*0.5#-gamma_s_2

      elif round(vert_nodes[2].Y,1)==round(vert_nodes[0].Y,1):
          theta_1_s=alfa_ang_s*0.5
          theta_2_s=beta_ang_s*0.5+gamma_s_2

      return theta_1_s, theta_2_s

    #Angulo punto inferior
    if vert_nodes[1].X<=vert_nodes[0].X:

      if round(vert_nodes[0].Y,1)==round(vert_nodes[2].Y,1) and round(vert_nodes[1].Y,1)==round(vert_nodes[3].Y,1):

          alfa_ang_i=abs(vec_01.AngleBetween(vec_02)*math.pi/180)
          beta_ang_i=abs(vec_10.AngleBetween(vec_13)*math.pi/180)
          gamma_i=vec_y.AngleBetween(vec_01)*math.pi/180
```

```
        theta_1_i=alfa_ang_i*0.5+gamma_i
        theta_2_i=beta_ang_i*0.5

        angi=[theta_1_i,theta_2_i]

      else:

        angi=ang_compi_y()

      #Puntos auxiliares para formar los triangulos y los trapecios
      #Se usan expresiones para X y Y global
      aux_point_i=Point.ByCoordinates((LY01-math.tan(angi[1])*LX01)/(math.tan(angi[0])+math.tan(angi[1]))+vert_nodes[0].X, \
        math.tan(angi[0])*((LY01-math.tan(angi[1])*LX01)/(math.tan(angi[0])+math.tan(angi[1])))+vert_nodes[0].Y, \
          vert_nodes[0].Z)

    else:

      if round(vert_nodes[0].Y,1)==round(vert_nodes[2].Y,1) and round(vert_nodes[1].Y,1)==round(vert_nodes[3].Y,1):

        alfa_ang_i=abs(vec_01.AngleBetween(vec_02)*math.pi/180)
        beta_ang_i=abs(vec_10.AngleBetween(vec_13)*math.pi/180)
        gamma_i=vec_y.AngleBetween(vec_01)*math.pi/180
        theta_1_i=alfa_ang_i*0.5
        theta_2_i=beta_ang_i*0.5+gamma_i

        angi=[theta_1_i,theta_2_i]

      else:

        angi=ang_compi_y()

      #Puntos auxiliares para formar los triangulos y los trapecios
      #Se usan expresiones para X y Y global
      aux_point_i=Point.ByCoordinates((LY01+math.tan(angi[1])*LX01)/(math.tan(angi[0])+math.tan(angi[1]))+vert_nodes[0].X, \
        math.tan(angi[0])*((LY01+math.tan(angi[1])*LX01)/(math.tan(angi[0])+math.tan(angi[1])))+vert_nodes[0].Y, \
          vert_nodes[0].Z)


    #Angulo punto superior
    if vert_nodes[2].X<=vert_nodes[3].X:

      if round(vert_nodes[0].Y,1)==round(vert_nodes[2].Y,1) and round(vert_nodes[1].Y,1)==round(vert_nodes[3].Y,1):

        alfa_ang_s=vec_20.AngleBetween(vec_23)*math.pi/180
        beta_ang_s=vec_31.AngleBetween(vec_32)*math.pi/180
        gamma_s=vec_y.AngleBetween(vec_23)*math.pi/180
        theta_1_s=alfa_ang_i*0.5+gamma_s
        theta_2_s=beta_ang_i*0.5

        angs=[theta_1_s,theta_2_s]

      else:

        angs=ang_comps_y()

      LY23=-LY23

      aux_point_s=Point.ByCoordinates((LY23+math.tan(angs[1])*LX23)/(math.tan(angs[0])+math.tan(angs[1]))+vert_nodes[2].X, \
        -math.tan(angs[0])*((LY23+math.tan(angs[1])*LX23)/(math.tan(angs[0])+math.tan(angs[1])))+vert_nodes[2].Y, \
          vert_nodes[2].Z)

    else:

      if round(vert_nodes[0].Y,1)==round(vert_nodes[2].Y,1) and round(vert_nodes[1].Y,1)==round(vert_nodes[3].Y,1):

        alfa_ang_s=vec_20.AngleBetween(vec_23)*math.pi/180
        beta_ang_s=vec_31.AngleBetween(vec_32)*math.pi/180
        gamma_s=vec_y.AngleBetween(vec_23)*math.pi/180
        theta_1_s=alfa_ang_i*0.5
        theta_2_s=beta_ang_i*0.5+gamma_s

        angs=[theta_1_s,theta_2_s]

      else:

        angs=ang_comps_y()

      LY23=-LY23

      aux_point_s=Point.ByCoordinates((LY23-math.tan(angs[1])*LX23)/(math.tan(angs[0])+math.tan(angs[1]))+vert_nodes[2].X, \
        -math.tan(angs[0])*((LY23-math.tan(angs[1])*LX23)/(math.tan(angs[0])+math.tan(angs[1])))+vert_nodes[2].Y, \
          vert_nodes[2].Z)



    slabs_vec[i].add_tri_points(aux_point_i)
    slabs_vec[i].add_tri_points(aux_point_s)

    #Se obtienen las lineas de cada triangulo y trapecio
    #Triangulo inferior
    line_aux_11=Line.ByStartPointEndPoint(vert_nodes[0],aux_point_i)
    line_aux_12=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[1])
    line_aux_13=Line.ByStartPointEndPoint(vert_nodes[1],vert_nodes[0])

    #Triangulo superior
    line_aux_21=Line.ByStartPointEndPoint(vert_nodes[2],aux_point_s)
    line_aux_22=Line.ByStartPointEndPoint(aux_point_s,vert_nodes[3])
    line_aux_23=Line.ByStartPointEndPoint(vert_nodes[3],vert_nodes[2])

    #Trapecio
    line_aux_31=Line.ByStartPointEndPoint(vert_nodes[1],vert_nodes[3])
```

```
    line_aux_32=Line.ByStartPointEndPoint(vert_nodes[3],aux_point_s)
    line_aux_33=Line.ByStartPointEndPoint(aux_point_s,aux_point_i)
    line_aux_34=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[1])

    #Trapecio
    line_aux_41=Line.ByStartPointEndPoint(vert_nodes[0],vert_nodes[2])
    line_aux_42=Line.ByStartPointEndPoint(vert_nodes[2],aux_point_s)
    line_aux_43=Line.ByStartPointEndPoint(aux_point_s,aux_point_i)
    line_aux_44=Line.ByStartPointEndPoint(aux_point_i,vert_nodes[0])


    curves_aux_1=[line_aux_11,line_aux_12,line_aux_13]
    curves_aux_2=[line_aux_21,line_aux_22,line_aux_23]
    curves_aux_3=[line_aux_31,line_aux_32,line_aux_33,line_aux_34]
    curves_aux_4=[line_aux_41,line_aux_42,line_aux_43,line_aux_44]



  elif (length_02*length_13==length_01*length_23):

    #Losa con lados iguales

    #Punto auxiliar que denota el centroide del cuadrado
    #Intersecci n de lineas 03 y 12
    aux_point_cent=[]

    line_auxc_03=Line.ByStartPointEndPoint(vert_nodes[0],vert_nodes[3])
    line_auxc_12=Line.ByStartPointEndPoint(vert_nodes[1],vert_nodes[2])

    aux_point_cent=line_auxc_03.Intersect(line_auxc_12)


    slabs_vec[i].add_tri_points(aux_point_cent[0])
    #slabs_vec[i].add_tri_points(aux_point_s)

    #Se obtienen las lineas de cada triangulo
    #Triangulo inferior
    line_aux_11=Line.ByStartPointEndPoint(vert_nodes[0],aux_point_cent[0])
    line_aux_12=Line.ByStartPointEndPoint(aux_point_cent[0],vert_nodes[2])
    line_aux_13=Line.ByStartPointEndPoint(vert_nodes[2],vert_nodes[0])

    #Triangulo superior
    line_aux_21=Line.ByStartPointEndPoint(vert_nodes[3],aux_point_cent[0])
    line_aux_22=Line.ByStartPointEndPoint(aux_point_cent[0],vert_nodes[1])
    line_aux_23=Line.ByStartPointEndPoint(vert_nodes[1],vert_nodes[3])

    #Triangulo izquierdo
    line_aux_31=Line.ByStartPointEndPoint(vert_nodes[0],vert_nodes[1])
    line_aux_32=Line.ByStartPointEndPoint(vert_nodes[1],aux_point_cent[0])
    line_aux_33=Line.ByStartPointEndPoint(aux_point_cent[0],vert_nodes[0])

    #Triangulo superior
    line_aux_41=Line.ByStartPointEndPoint(vert_nodes[2],aux_point_cent[0])
    line_aux_42=Line.ByStartPointEndPoint(aux_point_cent[0],vert_nodes[3])
    line_aux_43=Line.ByStartPointEndPoint(vert_nodes[3],vert_nodes[2])

    #Vectores que integran cada l nea generada #Hacer para cualquier forma
    curves_aux_1=[line_aux_11,line_aux_12,line_aux_13]
    curves_aux_2=[line_aux_21,line_aux_22,line_aux_23]
    curves_aux_3=[line_aux_31,line_aux_32,line_aux_33]
    curves_aux_4=[line_aux_41,line_aux_42,line_aux_43]


  return curves_aux_1,curves_aux_2,curves_aux_3,curves_aux_4

def slab_align(nodes):

  def min_max(nodes_min_max):
    #Identificacion de minimos
    x_min_vec=[]
    y_min_vec=[]

    for i in nodes_min_max:
      x_min_vec.append(i)
      y_min_vec.append(i)


    for i in range(nodes_min_max.Count):
      for j in range(nodes_min_max.Count):
        if x_min_vec[i].X<x_min_vec[j].X:

          x_min_aux=x_min_vec[j]
          x_min_vec[j]=x_min_vec[i]
          x_min_vec[i]=x_min_aux


        if y_min_vec[i].Y<y_min_vec[j].Y:

          y_min_aux=y_min_vec[j]
          y_min_vec[j]=y_min_vec[i]
          y_min_vec[i]=y_min_aux


    return x_min_vec,y_min_vec


  aligned_vec=[]
  nodes

  #primer identificaci n [0]
  min_max_vec=[]
  min_max_vec=min_max(nodes)
```

```
  #primer alineacion para losas [0]
  node_counter=nodes.Count
  i=0
  ##Organizaci n de nodos con respecto a apuntes -- Esta a prueba a n
  while(i<node_counter):
    #Para [0] se veifica con el segundo menor de X (ver ref. caso 3)
    if round(nodes[i].X,5)<=round(min_max_vec[0][1].X,5):
      #Se verifica con el menor segundo de Y
      if round(nodes[i].Y,5)<=round(min_max_vec[1][1].Y,5):

        aligned_vec.append(nodes[i])
        nodes.pop(i)
        node_counter=i

    i=i+1

  #segunda identificaci n [1]
  min_max_vec=[]
  min_max_vec=min_max(nodes)

  #segunda alineacion para losas [1]
  node_counter=nodes.Count
  i=0
  while(i<node_counter):
    if round(nodes[i].X,5)<=round(min_max_vec[0][0].X,5):
      aligned_vec.append(nodes[i])
      nodes.pop(i)
      node_counter=i

    i=i+1

  #tercer identificaci n [2]
  min_max_vec=[]
  min_max_vec=min_max(nodes)

  #tercer alineacion para losas [2]
  node_counter=nodes.Count
  i=0
  while(i<node_counter):
    if round(nodes[i].Y,5)<=round(min_max_vec[1][0].Y,5):
      aligned_vec.append(nodes[i])
      nodes.pop(i)
      node_counter=i

    i=i+1

  #ultimo valores
  aligned_vec.append(nodes[0])
  nodes.pop()

  return aligned_vec#, nodes_aux

def search_count_beams(beams, beams_index):

  #Se har n permutaciones con repeticion para tener las posibles "combinaciones" que tendr an
  #las barras entre el punto inicial y el final_f

  permut_2=[]

  search_flag=0
  for i in range(slabs_vec.Count):
    permut=[]
    for j in itertools.permutations(slabs_vec[i].surf_points_vec,2):

      permut.append(j)

    permut_2.append(permut)

    for j in range(beams.Count):

      search_flag=0
      for k in range(permut_2[i].Count):
        #if beams[j].StartPoint.IsAlmostEqualTo(permut[k][0]) and beams[j].EndPoint.IsAlmostEqualTo(permut[k][1]):
          if round(beams[j].StartPoint.X,5) == round(permut[k][0].X,5) and round(beams[j].StartPoint.Y,5) == round(permut[k][0].Y,5)\
            and round(beams[j].EndPoint.X,5) == round(permut[k][1].X,5) and round(beams[j].EndPoint.Y,5) == round(permut[k][1].Y,5):
              search_flag=1

        #elif beams[j].EndPoint.IsAlmostEqualTo(permut[k][0]) and beams[j].StartPoint.IsAlmostEqualTo(permut[k][1]):
          elif round(beams[j].EndPoint.X,5) == round(permut[k][0].X,5) and round(beams[j].EndPoint.Y,5) == round(permut[k][0].Y,5)\
            and round(beams[j].StartPoint.X,5) == round(permut[k][1].X,5) and round(beams[j].StartPoint.Y,5) == round(permut[k][1].Y,5):
              search_flag=1

      if search_flag==1:
        slabs_vec[i].add_cont_beams(beams_index[j])

  prueba_2=[]
  for s in range(slabs_vec.Count):
    prueba_2.append(slabs_vec[s].c_beams_vec)

  return prueba_2#,permut_2#, prueba_2

def load_assign(beams,index_frame_vec, slab_floor_vec):

  load_vec=[0.0]*index_frame_vec.Count

  for i in range(index_frame_vec.Count):
    for j in range(slab_floor_vec.Count):
      for k in range(slab_floor_vec[j].c_beams_vec.Count):
        if index_frame_vec[i] == slab_floor_vec[j].c_beams_vec[k]:
          aux_load=slab_floor_vec[j].tri_areas_vec[k]*load_factor/(beams[i].Length*100)
          load_vec[i]=(load_vec[i]+aux_load)

  return load_vec
```

A.8. Código: Análisis Estructural Utilizando MECA

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
import sys
sys.path.append(r'C:\Program Files (x86)\IronPython 2.7\Lib')
import os
import time
import subprocess

#The inputs to this node will be stored as a list in the IN variable.
dataEnteringNode = IN

"""///*** Main ***///"""

## Path Identification Start ##
appDataPath = os.getenv('APPDATA')

#Path para Dynamo Studio 2016
dynPath_1 = appDataPath + str('\Autodesk\Dynamo Studio\\0.9\\StructuralDynamo\\extra\\MECA')

#Path para Dynamo Open Source Version 0.9
dynPath_2 = appDataPath + str('\Dynamo\\0.9\\Packages\StructuralDynamo\\extra\\MECA')

if dynPath_1 not in sys.path:
  if dynPath_1 not in sys.path:
    sys.path.Add(dynPath_1)

if dynPath_2 not in sys.path:
  if dynPath_2 not in sys.path:
    sys.path.Add(dynPath_2)

SDPath_1_ver = appDataPath + str("\Autodesk\Dynamo Studio\\0.9\StructuralDynamo\extra\MECA\CatalogoCONCR.dat")
SDPath_2_ver = appDataPath + str("\Dynamo\\0.9\packages\StructuralDynamo\extra\MECA\CatalogoCONCR.dat")

if os.path.exists(SDPath_1_ver):
  SDPath=appDataPath + str("\Autodesk\Dynamo Studio\\0.9\StructuralDynamo\extra")
if os.path.exists(SDPath_2_ver):
  SDPath=appDataPath + str("\Dynamo\\0.9\packages\StructuralDynamo\extra")

dataEnteringNode = IN

## Path Identification Finish ##

global_time_ini=time.time()

today_str=time.strftime("%d_%m_%Y_")
time_str=time.strftime("%H_%M_%S")


appDataPath_folder = os.getenv('USERPROFILE')
dynpath_folder=appDataPath_folder+("\Documents\StructuralDynResults")


cwd_1 = dynpath_folder+str('\MECA\MECA_DYNAMO\Dynamo_')
file_url=str(cwd_1)
#file_name=file_url+today_str+time_str+(".txt")

folder = file_url+today_str+time_str
pre_file_name=folder+str("\Dynamo_")+today_str+time_str
file_name=pre_file_name+str(".dat")

os.makedirs(folder)
frames_data_in=IN[0][0][0]
loads_in=IN[0][2]
supports_restriction=IN[1]

archivo=MECA_data(file_name,frames_data_in,loads_in,supports_restriction)

## Analisis con MECA ##
if IN[2]==True:
  cwd = SDPath+str("\MECA\MECA_b.dat")
  MECA_bat=open(cwd,'w')
  cwd_2 = SDPath+str("\MECA\ ")
  MECA_bat.write(str("cd ")+cwd_2+str("\n"))
  MECA_bat.write("MECA_NOP15.exe")
  MECA_bat.close()

  cwd_2 = SDPath+str("\MECA\MECA_b")
  pre=cwd_2
  ext=".dat"

  cwd_2 = SDPath+str("\MECA\coman.dat")
  MECA_coman=open(cwd_2,'w')
  MECA_coman.write(folder+str("\Dynamo_")+today_str+time_str+str(".dat \n"))
  MECA_coman.write(folder+str("\Dynamo_")+today_str+time_str+str(".res"))
  MECA_coman.close()

  renamee=cwd
  pre, ext= os.path.splitext(renamee)
  os.rename(renamee, pre + ".bat")

  cwd_2 = SDPath+str("\MECA\MECA_b.bat")
  MECA_process=subprocess.Popen(cwd_2)

  MECA_process.wait()
  os.remove(cwd_2)



global_time_fin=time.time() - global_time_ini
conec_parameters=[frames_data_in,archivo]
#Assign your output to the OUT variable
OUT = conec_parameters,pre_file_name,archivo, global_time_fin

"""///*** Functions ***///"""



def MECA_data(file_name, frames_data, loads, restraint):

  MECAf=open(file_name,'w')


  #Comienza escritura de archivo
  MECAf.write('PreMECA_____________________________________= 1.0\n')
  MECAf.write('Proyecto_____________________________________= Dynamo\n')
  MECAf.write('Unidadeslongitud_____________________________= Cm,Kg,Seg\n')
```

```
MECAf.write('Unidades_fuerza_____= Cm,Kg,Seg\n')

n_nodes=0
#Conteo de nodos
for f in range(frames_data[1][1].Count):
    n_nodes=n_nodes+frames_data[1][1][f].Count

n_nodes=str(n_nodes)

n_bars=0
n_beams=0
#Conteo de vigas
for f in range(frames_data[1][2].Count):
    n_bars=n_bars+frames_data[1][2][f].Count
    #n_beams para saber cuantas cargas utilizar
    n_beams=n_beams+frames_data[1][2][f].Count

#Conteo de columnas
for f in range(frames_data[0][1].Count):
    n_bars=n_bars+frames_data[0][1][f].Count

n_bars=str(n_bars)

MECAf.write('Numero_de_nodos_____= ')
MECAf.write(n_nodes)
MECAf.write('\n')

MECAf.write('Numero_de_elementos_____= ')
MECAf.write(n_bars)
MECAf.write('\n')


#N mero de prescripciones ser n por el momento el n mero de nodos en el piso
n_presc=str(frames_data[1][1][0].Count)

MECAf.write('Numero_de_prescripciones_____= ')
MECAf.write(n_presc)
MECAf.write('\n')

MECAf.write('Numero_de_casos_de_carga_____= 1\n')
MECAf.write(' Tipo_de_estructura__art,_emp_o_comb____= 3\n')
MECAf.write('Numero_de_grados_de_libertad_por_nodo__= 6\n')

MECAf.write('Numero_de_materiales_____= ')
n_mat=frames_data[2][0].Count
MECAf.write(str(n_mat))
MECAf.write('\n')

MECAf.write('Escritura_dat_en_res_____0_o_1__= 1\n')
MECAf.write('Escritura_de_mat_de_rigid_____0_o_1__= 0\n')
MECAf.write('Numero_de_dimensiones_____2_o_3= 3\n')

MECAf.write('Numero_de_apoyos_elasticos_____= 0\n')
MECAf.write('Numero_de_apoyos_inclinados_____= 0\n')
MECAf.write(' Numero_de_familias___(optimizacion?)___= 1\n')
MECAf.write('Tipo_de_problema_____estatico_o_dinam__= 1\n')

#Definici n de barras
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\==========DEFINICION_DE_BARRAS==========/////\n')
MECAf.write('#No_material Nod_ini Nod_fin [RNod_aux] [RTipo_de_barra] [Rot_sobre_eje_prop]\n')
MECAf.write('#[ang_1;ang_2;ang_3]\n')

#La topologia de barras se ordena como se realizo la cadena, los dem s datos ser n fijos por el momento


#Topolog a de vigas
for f in range(frames_data[1][4].Count):
    for i in range(frames_data[1][4][f].Count):
        MECAf.write(str(frames_data[2][1][f][i][0]))
        MECAf.write('\t')
        MECAf.write(str(frames_data[1][4][f][i][0]+1))
        MECAf.write('\t')
        MECAf.write(str(frames_data[1][4][f][i][1]+1))
        MECAf.write('\t')
        #Se fijaran los siguientes datos por el momento
        MECAf.write('0\t')
        MECAf.write('2\t')
        MECAf.write('0.0\n')

#Topolog a de columnas
for f in range(frames_data[0][3].Count):
    for i in range(frames_data[0][3][f].Count):
        MECAf.write(str(frames_data[2][2][f][i][0]))
        MECAf.write('\t')
        MECAf.write(str(frames_data[0][3][f][i][0]+1))
        MECAf.write('\t')
        MECAf.write(str(frames_data[0][3][f][i][1]+1))
        MECAf.write('\t')
        #Se fijaran los siguientes datos por el momento
        MECAf.write('0\t')
        MECAf.write('2\t')
        MECAf.write('0.0\n')



#Coordenadas de nodos
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\==========COORDENADAS_DE_NODOS==========/////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#x y z\n')
```

```
#Conteo de nodos
for f in range(frames_data[1][1].Count):
  for i in range(frames_data[1][1][f].Count):

    #Se multiplica por 100 pensando que estamos trabajando de metros a cm
    MECAf.write(str(format(100*frames_data[1][1][f][i].X,'.6f')))
    MECAf.write("\t")
    MECAf.write(str(format(100*frames_data[1][1][f][i].Y,'.6f')))
    MECAf.write("\t")
    MECAf.write(str(format(100*frames_data[1][1][f][i].Z,'.6f')))
    MECAf.write("\n")

#Prescripciones
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\=================RESTRICCIONES=================////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#nodo x y [z] Rx [Ry] [Rz] Presc_x Presc_y [Presc_z] Presc_Rx [Presc_Ry]\n')
MECAf.write('#[Presc_Rz]\n')

#Se tomaron las prescripciones como la restricciones en la base de la edificaci n
#Si supports_restriction=true, se toman empotrados los apoyos, de lo contrario ser n articulados
#adem s se dibujaran los apoyos representativos

supports=[]

#Empotrados
if restraint==True:
  for i in range(frames_data[1][0][0].Count):

    MECAf.write(str(frames_data[1][0][0][i]+1))
    MECAf.write('\t')
    MECAf.write('1\t1\t1\t1\t1\t1\t0.0\t0.0\t0.0\t0.0\t0.0\t0.0\t0.0\n')

  #Representaci n de empotramiento con cubos
  for i in range(frames_data[1][1][0].Count):
    h=0.4 #altura del cubo
    w=0.8 #Dimensiones
    auxiliar_point=Point.ByCoordinates(frames_data[1][1][0][i].X,frames_data[1][1][0][i].Y,frames_data[1][1][0][i].Z-h/2)
    supports.append(Cuboid.ByLengths(auxiliar_point,w,w,h))

#Articulados
else:
  for i in range(frames_data[1][0][0].Count):

    MECAf.write(str(frames_data[1][0][0][i]+1))
    MECAf.write('\t')
    MECAf.write('1\t1\t1\t0\t0\t0\t0.0\t0.0\t0.0\t0.0\t0.0\t0.0\t0.0\n')

  #Representaci n de articulaciones con conos
  for i in range(frames_data[1][1][0].Count):
    r=0.4 #radio
    w=0.8 #altura
    auxiliar_point=Point.ByCoordinates(frames_data[1][1][0][i].X,frames_data[1][1][0][i].Y,frames_data[1][1][0][i].Z-w)
    supports.append(Cone.ByPointsRadius(auxiliar_point,frames_data[1][1][0][i],r))

#Definicion de materiales
MECAf.write('\n')
MECAf.write('#\\\\=================DEFINICION_DE_MATERIALES=================////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#Modulo_Elastico Area Inerc_x [Inerc_y] [Inerc_Polar] [Modulo_de_Corte]\n')
MECAf.write('#Peso_Esp. Tipo_Material Nombre_Cat Num_Sec\n')

for i in range(n_mat):
  MECAf.write('0.0 0.0 0.0 0.0 0.0 0.0 ')
  if int(frames_data[2][0][i][0])==3:
    MECAf.write('0.0024 ')
  else:
    MECAf.write('0.007850 ')

  MECAf.write(str(frames_data[2][0][i][0]))
  MECAf.write(' ')
  MECAf.write(str(frames_data[2][0][i][1]))
  MECAf.write('.dat ')
  MECAf.write(str(frames_data[2][0][i][2]))
  MECAf.write('\n')

#Apoyos El sticos
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\=================APOYOS_ELASTICOS=================////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#Nodo x y [z] Rx [Ry] [Rz]\n')

#Apoyos inclinados
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\=================APOYOS_INCLINADOS=================////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#Nodo Inclinacion\n')

#Casos de carga
MECAf.write('\n')
MECAf.write('#\\\\\\\\\\=================CASOS_DE_CARGA=================////\n') #Se le da un mayor numero de textbackslash debido a que \\ lo toma como \ porque piensa que es comando de escritura
MECAf.write('#Titulo\n')
MECAf.write('#NF_nodos NF_punt [NFpunt_y] NF_dist [NF_dist_y] [NF_triang] [NF_parabol]\n')
MECAf.write('#F_peso_prop\n\n')
```

```
311  #Por ahora se limitara a peso propio para pruebas del analisis
312  MECAf.write('Cargas_Distribuidas_No_Peso_propio\n')
313  MECAf.write('0\t0\t0\t')
314  n_beam_loads=str(n_beams)
315  MECAf.write(str(n_beam_loads))
316  MECAf.write('\t0\t0\n')
317
318
319  for f in range(frames_data[1][2].Count):
320      for i in range(frames_data[1][2][f].Count):
321          MECAf.write(str(frames_data[1][2][f][i]+1))
322          MECAf.write('\t-')
323          #Peso del concreto kg/cm3
324          w_con=.24
325          MECAf.write(str(loads[f][i]))
326          MECAf.write('\n')
327
328  MECAf.close()
329  return supports
```

A.9. Código: Resultados de Eficiencia Mecánica

```
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN
import sys
sys.path.append(r'C:\Program Files (x86)\IronPython 2.7\Lib')
import os
import time
import subprocess


"""///*** Main ***///"""

frames_data=IN[0][0][0]
cols_data=frames_data[0]
beams_data=frames_data[1]
MECA_results_file=IN[0][1]
despl=read_res(MECA_results_file, cols_data,beams_data)

supports=IN[0][2]
time_an=IN[0][3]
#Assign your output to the OUT variable.
OUT = despl,supports, time_an

"""///*** Functions ***///"""


def read_res(file_name, cols_data, beams_data):


  file_name_res=file_name+(".res")
  MECA_res=open(file_name_res)

  for i in range(3):
    nu_text=MECA_res.readline()

  n_elements=MECA_res.readline()
  n_elements_v=[]
  n_elements_v=n_elements.split()

  n_nodes=int(n_elements_v[0])
  n_frames=int(n_elements_v[1])
  n_gnodes=int(n_elements_v[2])
  n_ccarga=int(n_elements_v[3])
  n_tipoes=int(n_elements_v[4])
  n_gnoden=int(n_elements_v[5])
  n_mat=int(n_elements_v[6])
  #Por el momento se omitiran los valores restantes

  for i in range(n_frames):
    nu_text=MECA_res.readline()
  for i in range(n_nodes):
    nu_text=MECA_res.readline()
  for i in range(n_gnodes):
    nu_text=MECA_res.readline()
  for i in range(n_mat+1):
    nu_text=MECA_res.readline()

  n_loads=MECA_res.readline()
  n_loads_v=[]
  n_loads_v=n_loads.split()

  n_NF_nodes=int(n_loads_v[0])
  n_NF_pun=int(n_loads_v[1])
  n_NF_pun_y=int(n_loads_v[2])
  n_dist=int(n_loads_v[3])
  n_NF_tri=int(n_loads_v[4])
  n_self_w=int(n_loads_v[5])

  for i in range(n_NF_nodes):
    nu_text=MECA_res.readline()
  for i in range(n_NF_pun):
    nu_text=MECA_res.readline()
  for i in range(n_NF_pun_y):
    nu_text=MECA_res.readline()
  for i in range(n_dist):
    nu_text=MECA_res.readline()
  for i in range(n_NF_tri):
    nu_text=MECA_res.readline()

  for i in range(n_nodes):
    nu_text=MECA_res.readline()


  node_desp_num=[]

  factor=50

  for i in range(n_nodes):
    node_desp_split=[]
    node_desp_split_aux=[]
    node_desp_aux=[]
    node_desp_let=MECA_res.readline()



  #Desplazamiento de nodos
  desp_nodes=[]
  nodes_ND=[]
  for i in range(beams_data[1].Count):
    desp_nodes_aux=[]
    desp_nodes_aux_ND=[]
    for j in range(beams_data[1][i].Count):
      desp_nodes_aux.append(Point.ByCoordinates(beams_data[1][i][j].X,beams_data[1][i][j].Y,beams_data[1][i][j].Z))
      desp_nodes_aux_ND.append(beams_data[1][i][j])
    desp_nodes.append(desp_nodes_aux)
    nodes_ND.append(desp_nodes_aux_ND)

  for i in range(node_desp_num.Count):
    for f in range(beams_data[0].Count):
      for k in range(beams_data[0][f].Count):
        if i == beams_data[0][f][k]:
          desp_nodes[f][k]=desp_nodes[f][k].Translate(node_desp_num[i][0],node_desp_num[i][1],node_desp_num[i][2])

  #creacion de lineas
  nodes_index=[]
  desp_nodes_flat=[]
  nodes_ND_flat=[]
  for f in range(beams_data[0].Count):
    for i in range(beams_data[0][f].Count):
      nodes_index.append(beams_data[0][f][i])
      desp_nodes_flat.append(desp_nodes[f][i])
      nodes_ND_flat.append(nodes_ND[f][i])

  #vigas
  frame_index=[]
  for f in range(beams_data[4].Count):
    for i in range(beams_data[4][f].Count):
      frame_index.append(beams_data[4][f][i])

  #Columnas

  for f in range(cols_data[3].Count):
```

```
    for i in range(cols_data[3][f].Count):
      frame_index.append(cols_data[3][f][i])


  frame_lines=[]
  frame_lines_ND=[]
  for i in range(frame_index.Count):
    for j in range(nodes_index.Count):
      if frame_index[i][0]==nodes_index[j]:
        n_aux_1=desp_nodes_flat[j]
        n_aux_1_ND=nodes_ND_flat[j]
      if frame_index[i][1]==nodes_index[j]:
        n_aux_2=desp_nodes_flat[j]
        n_aux_2_ND=nodes_ND_flat[j]
    frame_lines.append(Line.ByStartPointEndPoint(n_aux_1, n_aux_2))
    frame_lines_ND.append(Line.ByStartPointEndPoint(n_aux_1_ND, n_aux_2_ND))


  ##Eficiencias

  #reacciones en apoyos
  for i in range(n_gnodes):
    nu_text=MECA_res.readline()

  #fuerzas en barra
  for i in range(n_frames):
    nu_text=MECA_res.readline()

  #eficiencias
  frame_efi_v=[]
  messages=[]
  for i in range(n_frames):
    frame_efi_split=[]
    frame_efi_split_aux=[]
    frame_efi_aux=[]
    frame_efi_let=MECA_res.readline()
    frame_efi_split=frame_efi_let.split("=")
    efi_value=float(frame_efi_split[2])
    if efi_value>=1:
      frame_efi_v.append(-1)
      messages.append("Barra:"+str(i+1)+" No Resiste")
    elif efi_value>=0:
      frame_efi_v.append(efi_value)


  MECA_res.close()

  return frame_efi_v, frame_lines_ND, messages
```

Apéndice B

Structural Dynamo: Manual Básico de Usuario

En este apartado se describe el funcionamiento de ***Structural Dynamo*** de manera sencilla mediante un ejemplo de aplicación. La primer sección hace referencia la instalación de la paquetería necesaria y posteriormente en la segunda sección se desarrollará el análisis de una edificación en base a un sólido rectangular.

B.1. Instalación Local

Dynamo funciona mediante una basta librería de geometría que permite a los usuarios crear redes DAG de manera que pueden realizar programación visual fácilmente. Además se provee la oportunidad de utilizar programas visuales desarrollados por usuarios, en forma de "paqueterías". La forma habitual de instalar una paquetería es mediante una conexión a internet y seleccionando la opción "packages". Sin embargo, para este trabajo se explicará como instalar ***Structural Dynamo*** de manera local.

Lo primero es asegurar que **Dynamo** esta correctamente instalado, y se debe de tener en cuenta que los módulos probados en este trabajo se desarrollaron utilizando la versión **Dynamo Studio 2016** y la versión Open Source. Para la versión de **Dynamo Studio 2016** se deberá buscar la siguiente ruta en la computadora: `C:/Users/USERNAME/AppData/Roaming/Autodesk/Dynamo Studio/0.9`[1] En esa carpeta se colocará la librería ***Structural Dynamo*** y cualquier otra que se desee instalar. Para asegurar que se encuentra correctamente instalado, ahora se iniciará **Dynamo** y en la biblioteca se buscará en las pestañas la denominada ***Structural Dynamo*** tal como se muestra en la siguiente figura:

[1] USERNAME se refiere al nombre de usuario del equipo en donde está instalado el software.

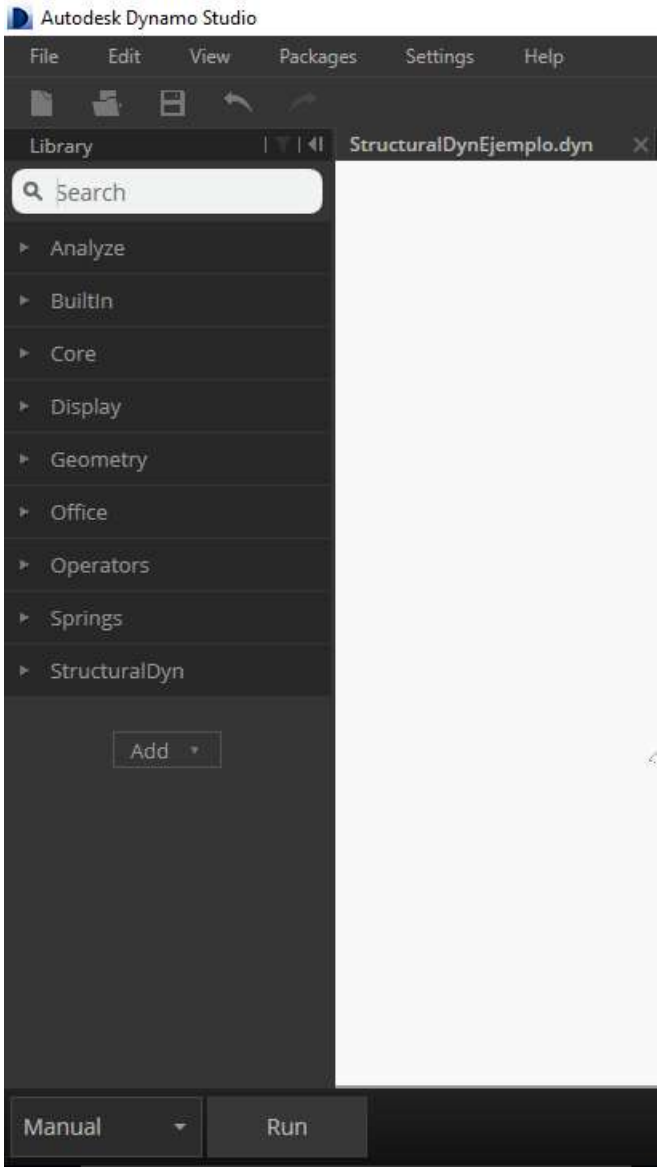


Figura B.1: Verificación de Instalación. Librería del software **Dynamo**, se muestra al final de la lista que se encuentra ***Structural Dynamo***, por lo tanto la instalación fue correcta.

B.2. Ejemplo Práctico: Análisis de una Edificación Rectangular

La intención de este apartado es mostrar lo sencillo que es hacer programación visual y además realizar un análisis estructural mediante los módulos desarrollados en este trabajo. Por ello, se comenzará desde la ventana de nuevo documento de **Dynamo**.

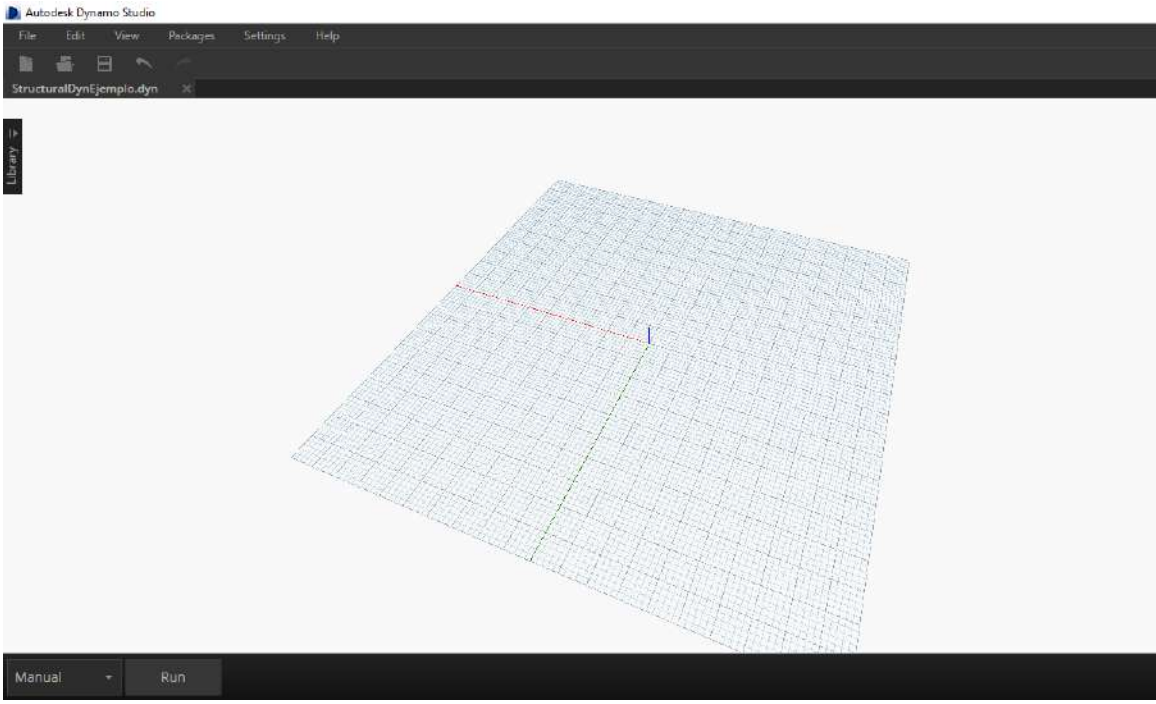


Figura B.2: Ventana Principal Dynamo. Se muestra el espacio donde se realiza la programación visual para generar geometría en el denominado "canvas".

Posteriormente se revisará la biblioteca de geometría que **Dynamo**tiene incorporada, en donde se buscará el paquete ***Structural Dynamo***:

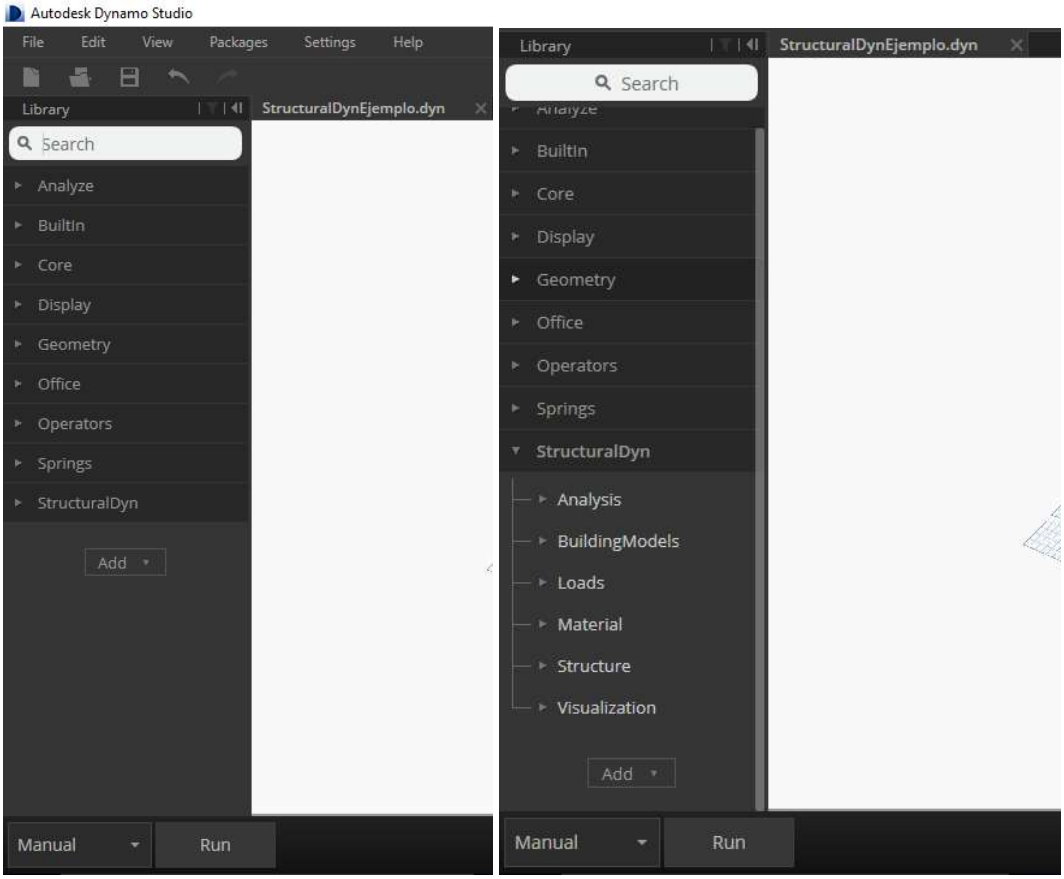


Figura B.3: Biblioteca Dynamo. Opciones para generar un programa visual en **Dynamo**. Se muestra el contenido del paquete ***Structural Dynamo***.

Para el primer paso, será necesario generar el modelo sólido tridimensional que describe la geometría conceptual de la edificación, para ello se utilizará la pestaña "BuildingModels" en donde se tienen pre-cargados algunos sólidos. La opción a elegir en este ejemplo será el edificio "Rectangular":

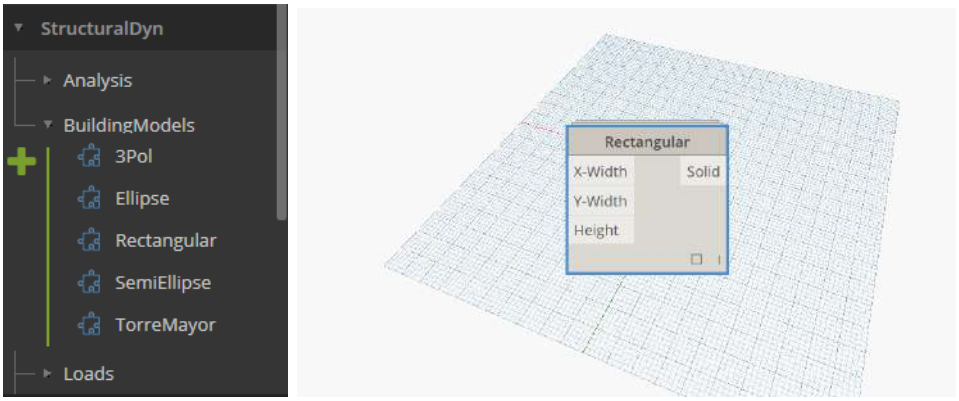


Figura B.4: Sólido Tridimensional. Selección de modelo de edificación rectangular.

Como podemos observar se necesita de tres parámetros para generar el sólido, "X, Y - Width" y "Height", por lo que se utilizará un nodo denominado "Number Slider" que facilita el cambio de valores numéricos:

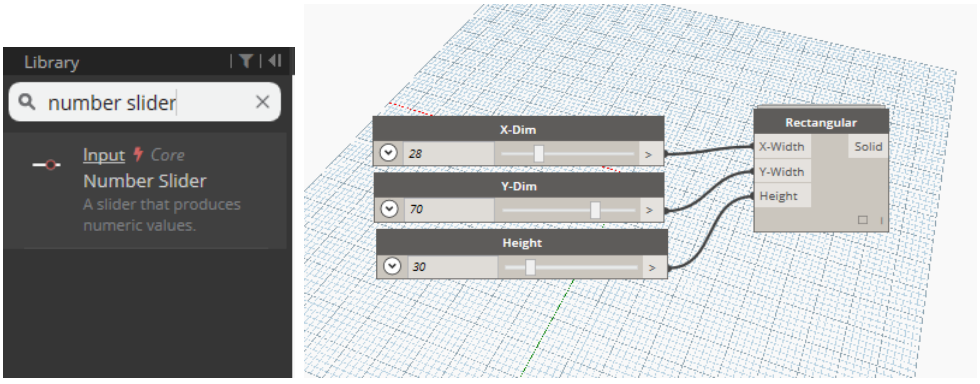


Figura B.5: Sólido Tridimensional. Parámetros para generar el sólido tridimensional.

Debido a que en la medida que vamos agregando nodos y haciendo conexiones se complica la manipulación de nuestro programa, **Dynamo**cuenta con una opción para crear grupos por colores y organizar cada fase del programa. Solo hay que seleccionar los nodos y dar click derecho para activar esta opción.

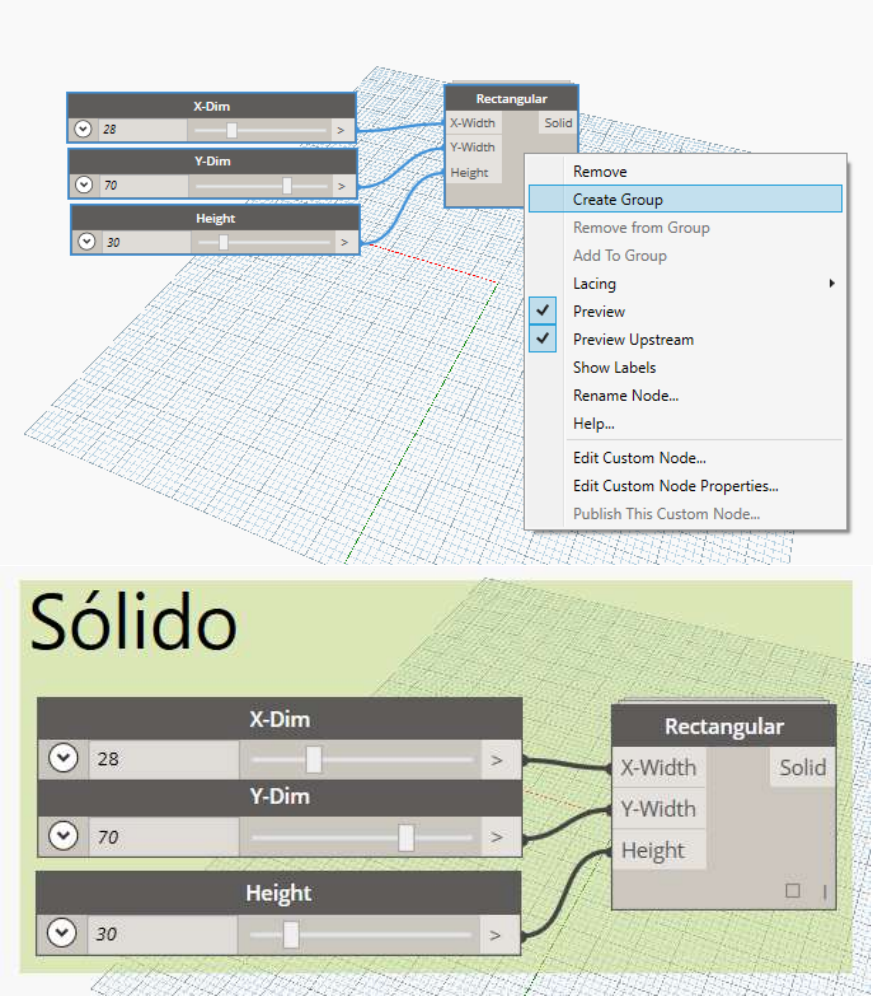


Figura B.6: Sólido Tridimensional. Creación de grupos por color.

Ahora se ejecutará el código al dar click en "Run" en la parte inferior izquierda de la ventana, lo que generará el sólido con las dimensiones asignadas:

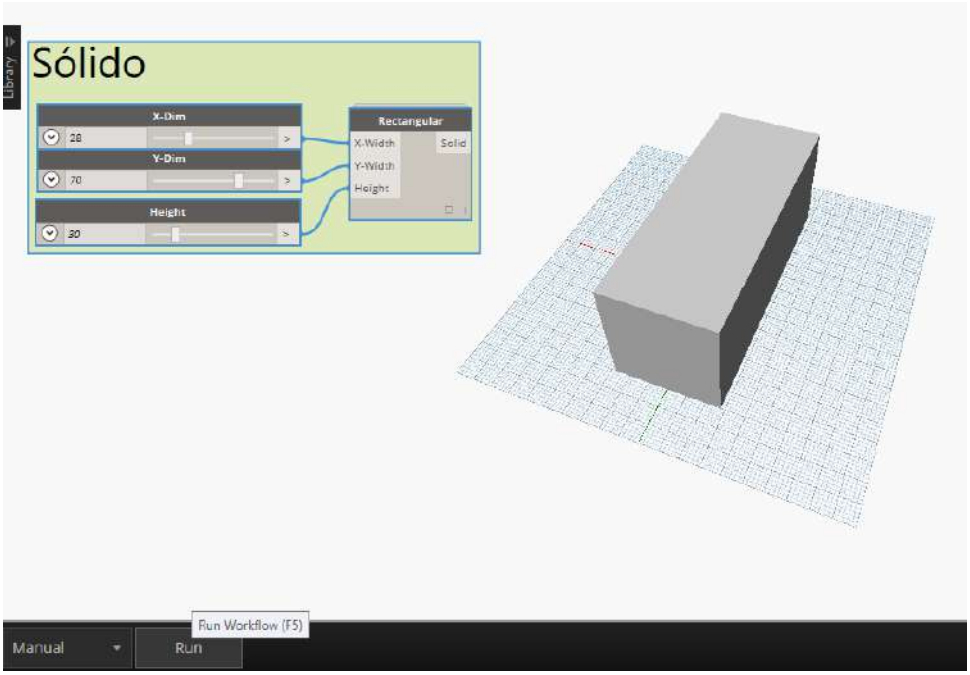


Figura B.7: Sólido Tridimensional. Generación de sólido al ejecutar el programa visual.

A un lado de "Run", se encuentra un botón de opción que dice "Manual". Esto quiere decir que cada vez que se quiera ejecutar el programa visual habrá que dar click en "Run". Pero si queremos un flujo más dinámico, se puede cambiar la opción a "Automatic", de esta manera al realizar cualquier cambio en parámetros, el modelo se actualizará automáticamente, como se muestra a continuación:

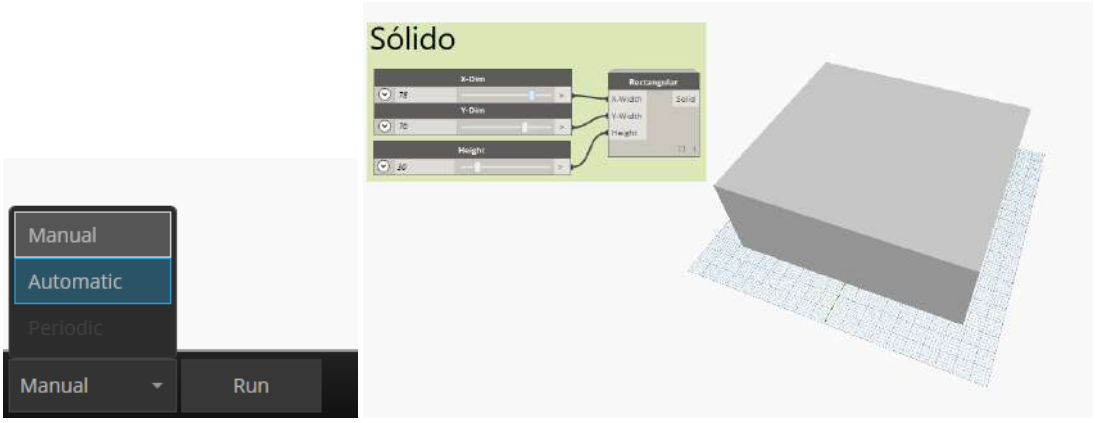


Figura B.8: Sólido Tridimensional. Opción "Automatic" para generación rápida de modelos.

Para este ejemplo se utilizará un modelo rectangular con dimensiones de sólido $20 - 20 - 15$:

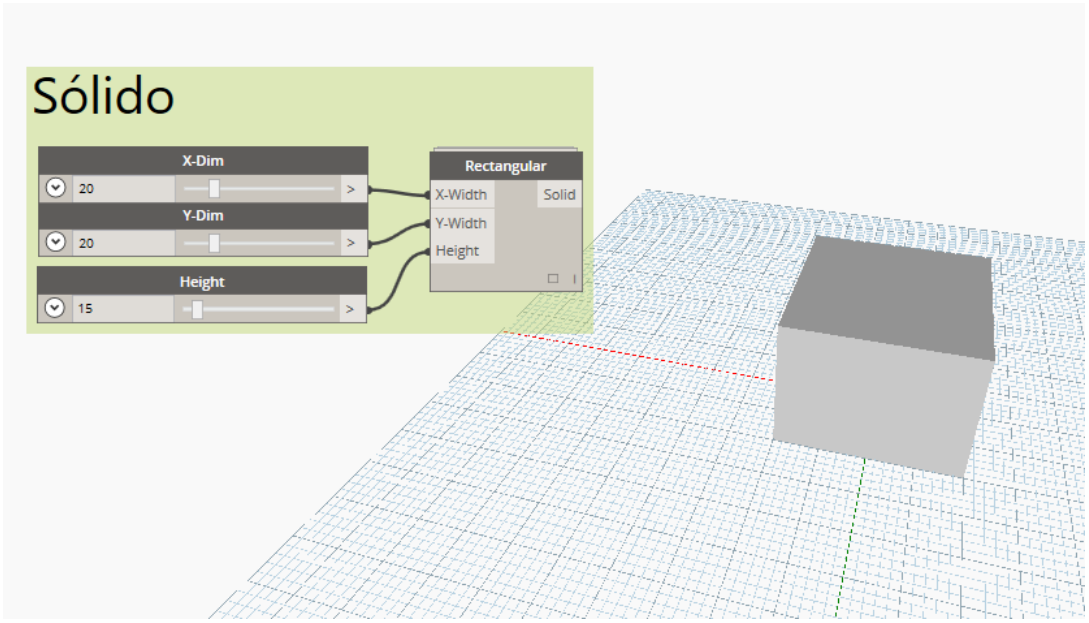


Figura B.9: Sólido Tridimensional. Sólido a estructurar.

El siguiente paso es utilizar la función ***Grid*** para proponer una configuración estructural:

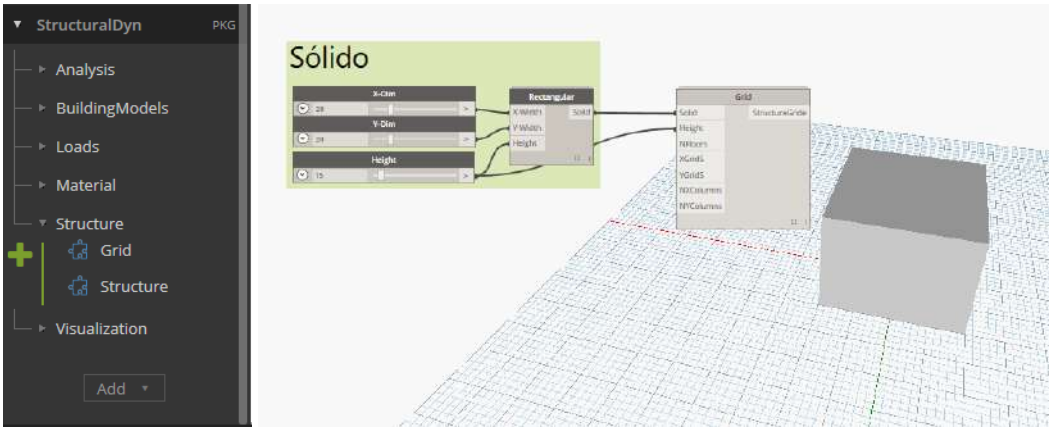


Figura B.10: Retícula de Estructuración. Se añade el nodo ***Grid*** para proponer una estructuración.

Como podemos observar, esta función necesita de más parámetros. De hecho esta es la fase donde el usuario tendrá mayor interacción con todo el análisis de estructuras, ya que es aquí donde se propone la configuración estructural a utilizar. Por ello se utilizarán los siguientes parámetros:

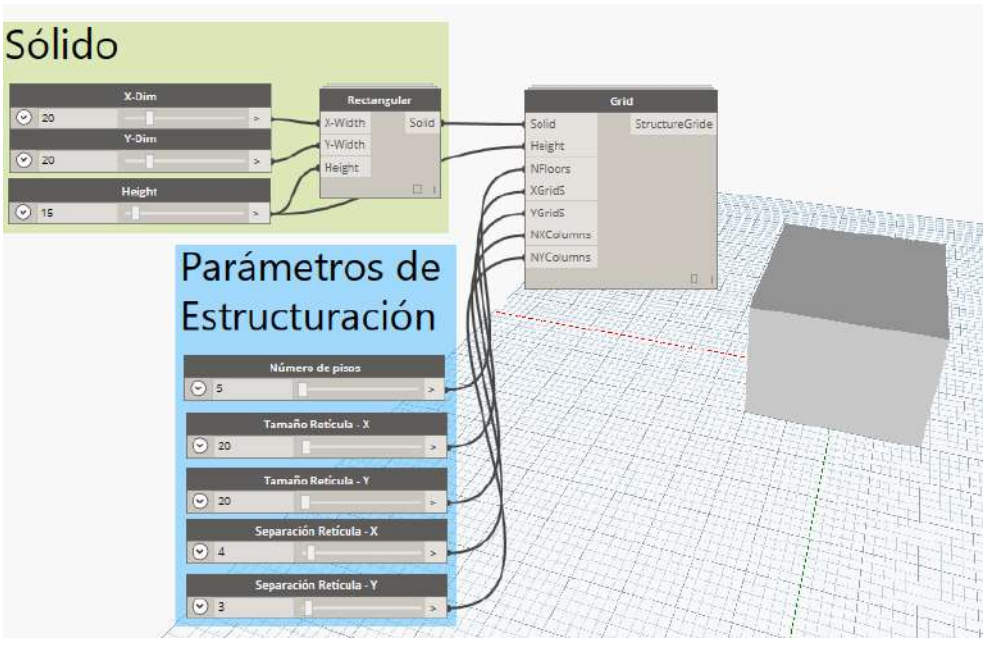


Figura B.11: Retícula de Estructuración. Parámetros de estructuración.

Ahora, para visualizar la retícula de estructuración será necesario utilizar el nodo que se encuentra en la pestaña de opciones de visualización "GridView", al

dar click derecho se activa "preview". Hay que tener en cuenta que el nodo ***Grid*** deberá tener desactivada esta opción, de lo contrario no se podrá observar la información necesaria:

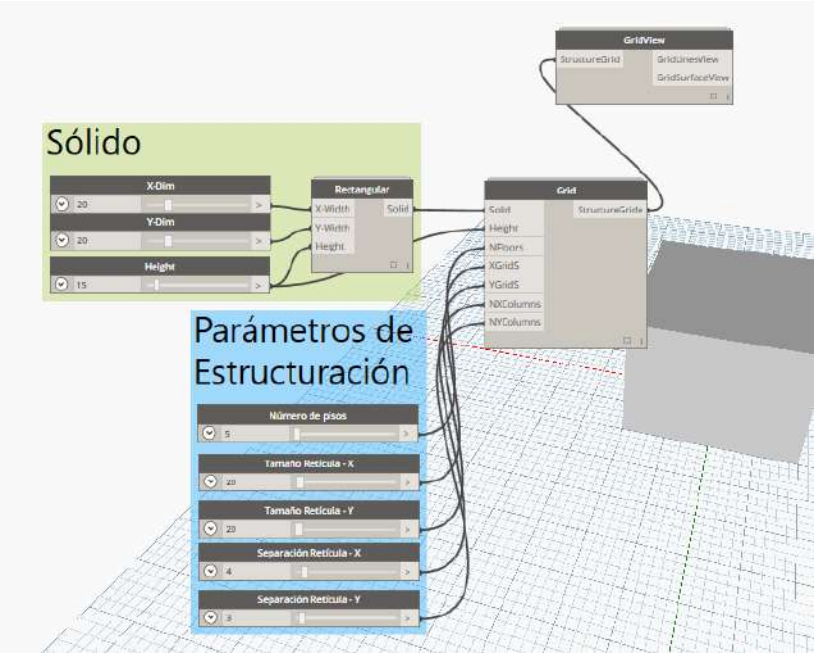


Figura B.12: Retícula de Estructuración. Visualización de ***Grid*** .

Como podemos observar, en este caso la "caja verde" tiene exactamente las dimensiones que nuestro sólido, además que se pueden ver las barras a utilizar en el modelo estructural.

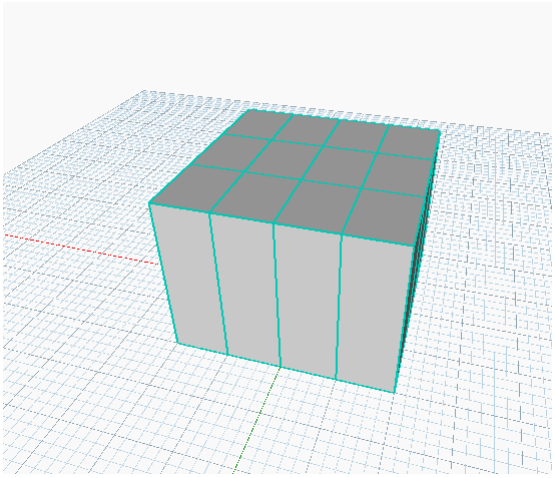


Figura B.13: Retícula de Estructuración. Visualización de ***Grid*** .

Retomando la imagen B.10, ahora seleccionamos la opción ***Structure***, la cual es la necesaria para obtener la estructura exterior e inferior, además de organizar la información estructural geométrica.

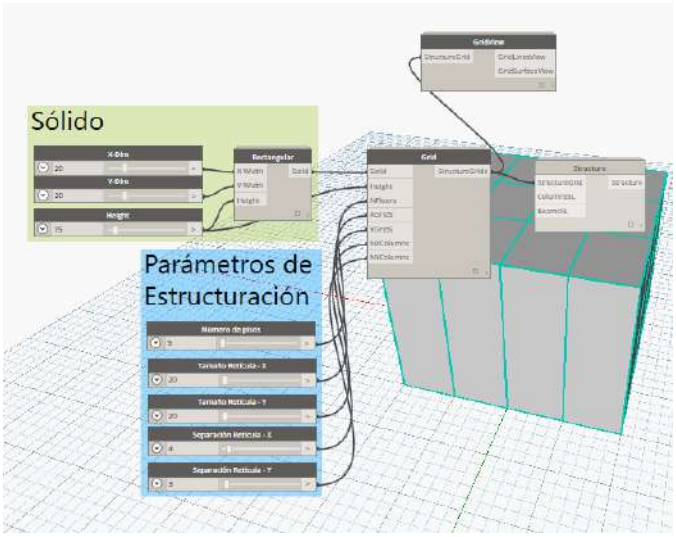


Figura B.14: Generación de Estructura. Se añade el nodo ***Structure***.

Pero como podemos observar hay dos parámetros faltantes, estos corresponden a los materiales usados en vigas y columnas del catálogo del software de análisis utilizado **"MECA"**. Se utilizaron en ambos elementos, el catálogo de concreto y las secciones mostradas, ya que se puede tener diferentes secciones por nivel de la edificación, éstos tienen que incorporarse a una lista:

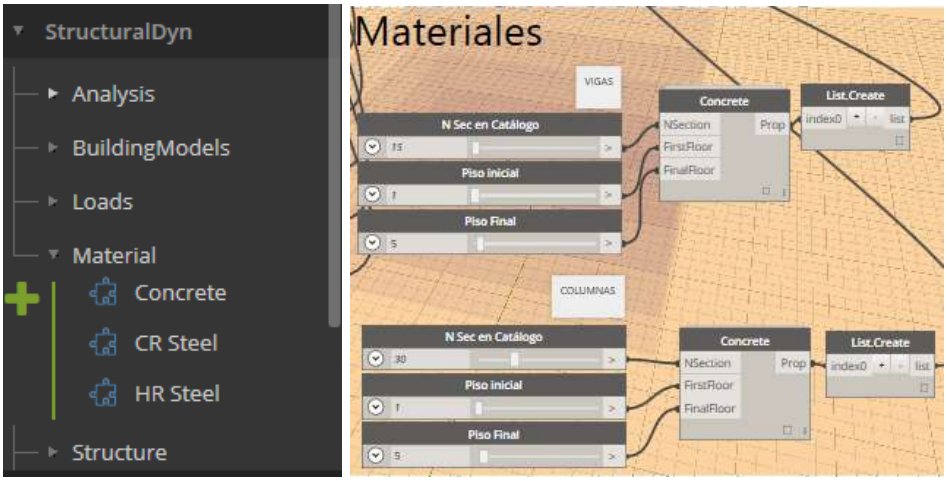


Figura B.15: Generación de Estructura. Material y sección del catálogo de **"MECA"**.

Se añade ahora el nodo de visualización "StructuralElementsView" y se muestra como se hace la conexión de materiales al módulo ***Structure***. Al finalizar este punto la rutina será un poco más lenta, esperar a que termine de ejecutarse el código.

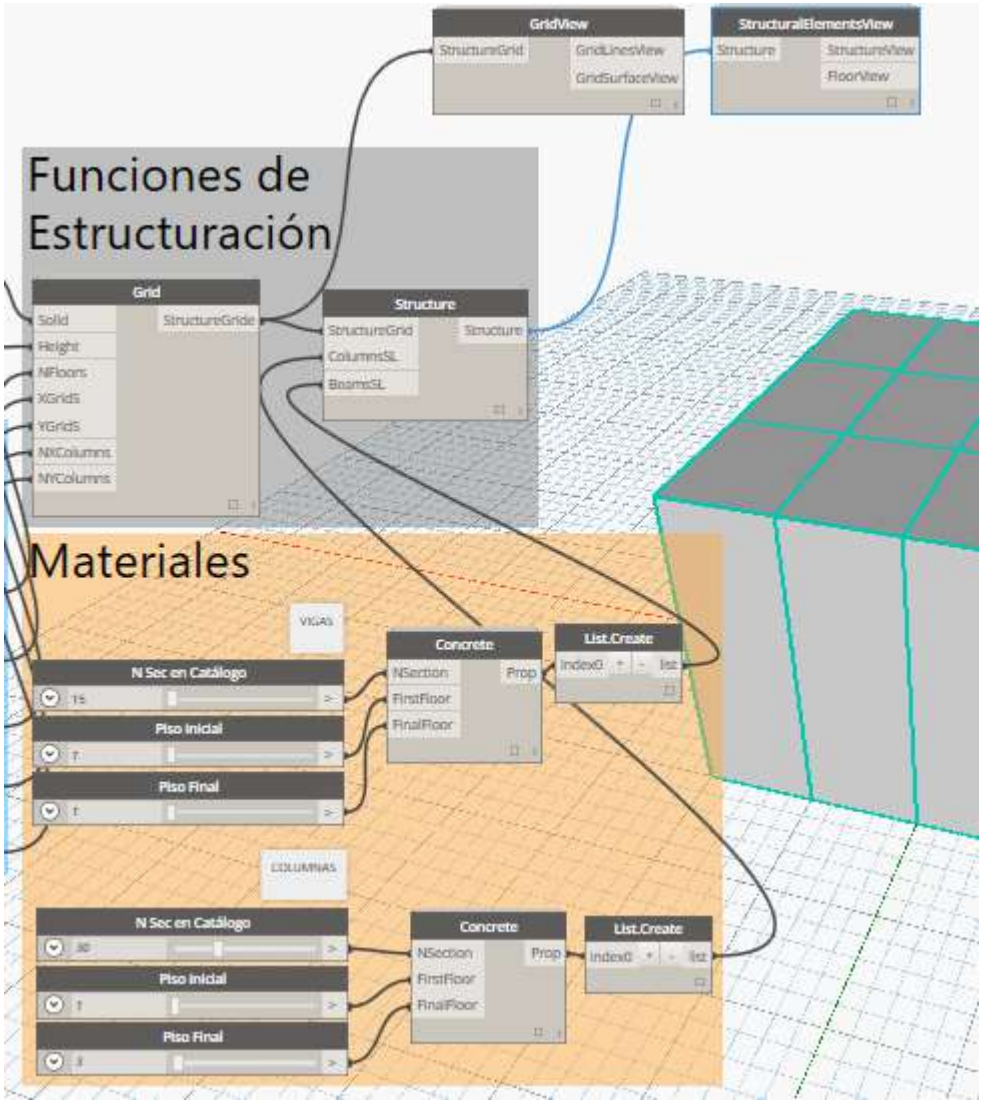


Figura B.16: Generación de Estructura. Conexión de materiales y visualización de elementos estructurales.

Al apagar la opción "preview" del sólido ***Rectangular*** y ***Grid*** y encender la correspondiente a los elementos estructurales, deberá aparecer una imagen como la siguiente:

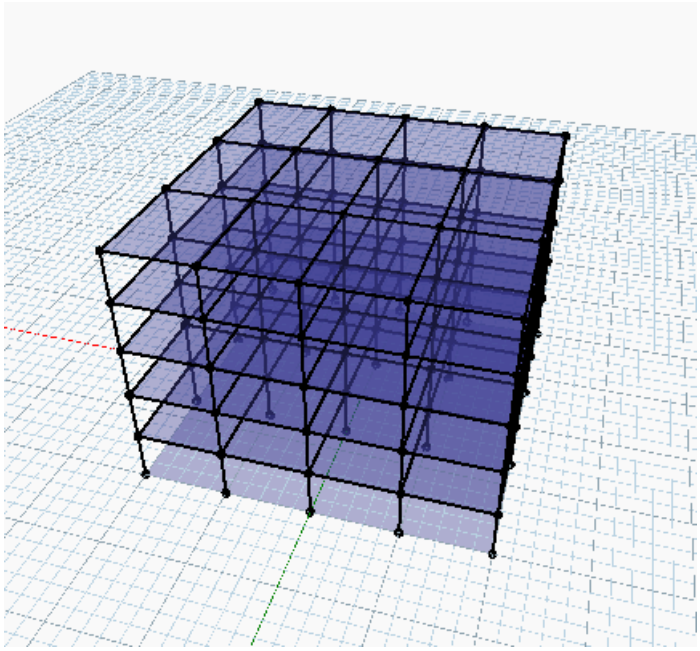


Figura B.17: Generación de Estructura. Visualización de elementos estructurales.

La siguiente fase es la asignación de fuerzas gravitacionales a utilizar en el modelo estructural, para ello se utilizará el módulo ***SlabLoads*** del paquete ***Structural Dynamo***y se conectará como se muestra en la imagen siguiente. Para tener un "script visual" se da doble click en la pantalla y se podrá escribir la carga a utilizar, de cualquier forma se puede usar "NumberSlider" también, pero la forma de entrada es mediante una lista.

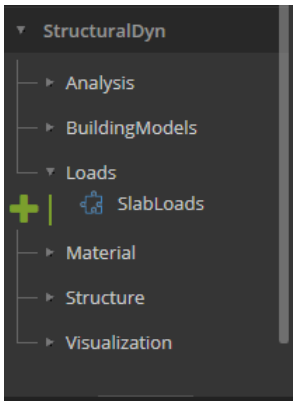


Figura B.18: Distribución de Cargas. Se añade el módulo ***SlabLoads***.

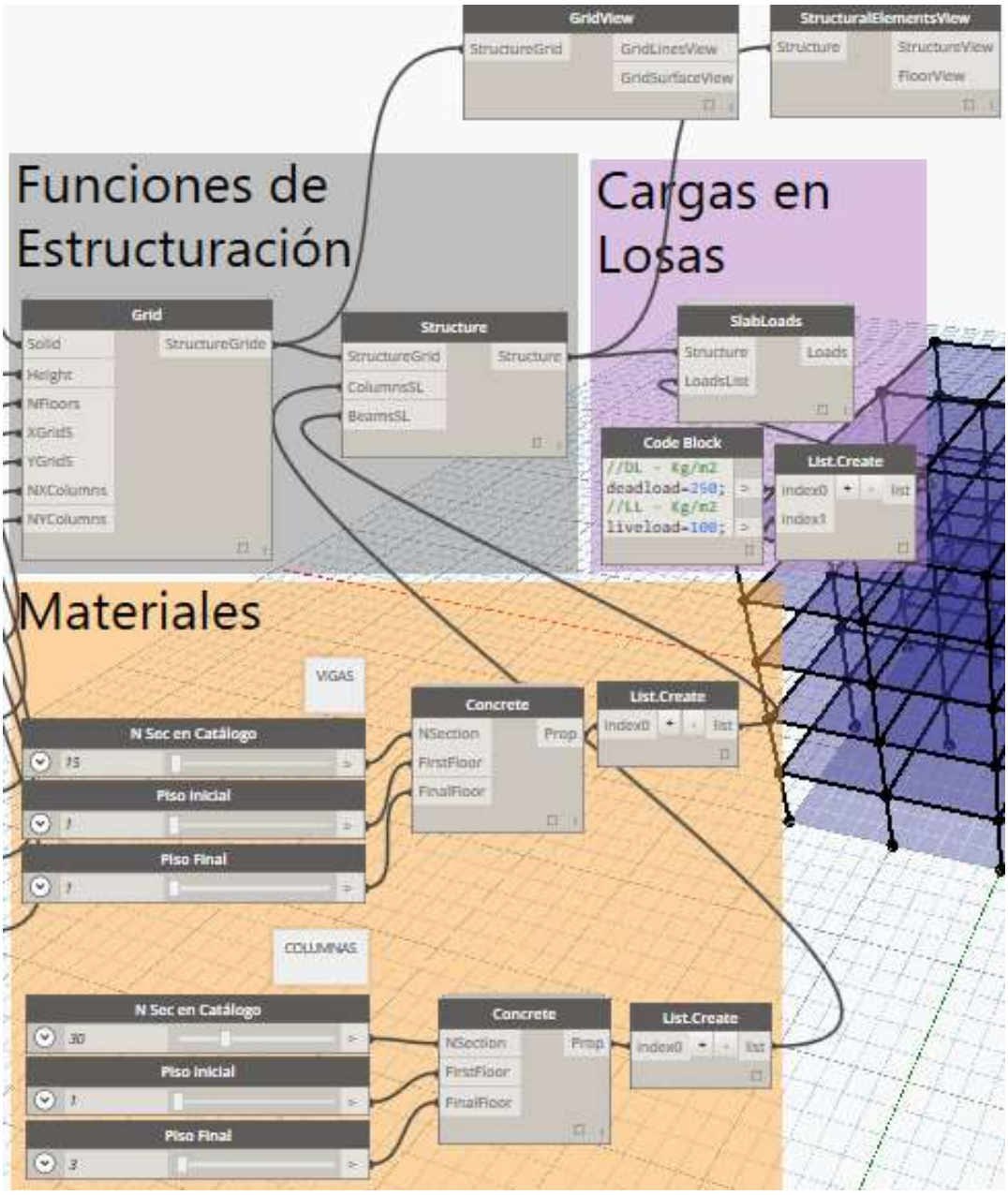


Figura B.19: Distribución de Cargas. Se muestra la conexión correcta para la asignación de cargas mediante ***SlabLoads***.

Si todo esta conectado correctamente, al ejecutar el código visual el resultado será el siguiente. Hágase notar que se está usando un nuevo nodo de visualización denominado "SlabLoadDistributionView":

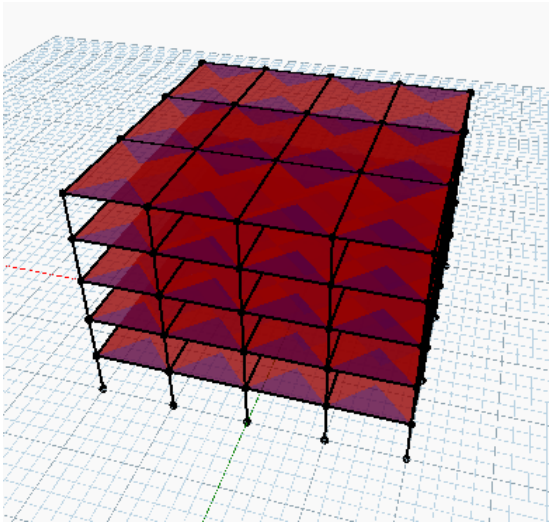


Figura B.20: Distribución de Cargas. Resultado del módulo ***SlabLoads***.

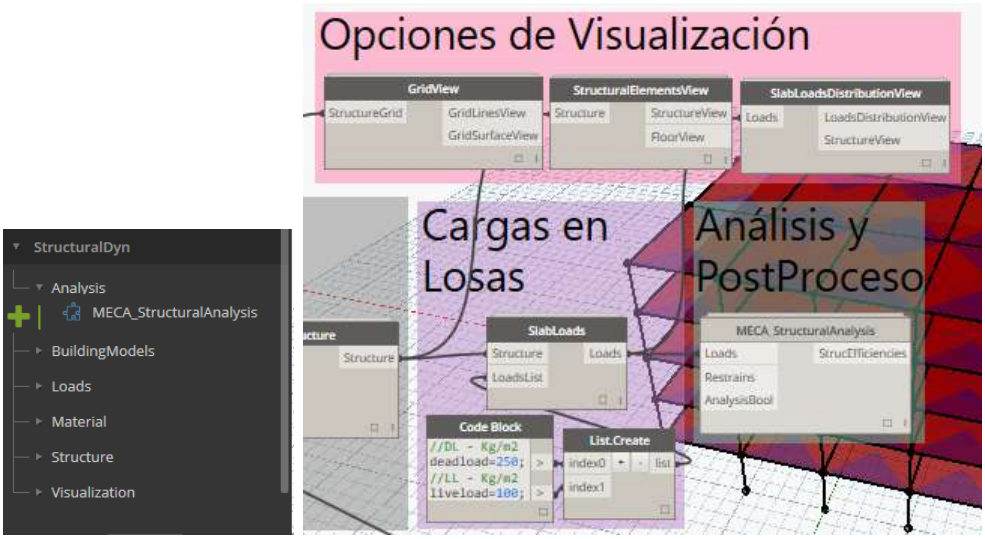


Figura B.21: Análisis Estructural. Se añade y se conecta el módulo ***MECA_StructuralAnalysis***.

El módulo ***MECA_StructuralAnalysis*** requiere de dos "booleanos" para ejecutarse, por lo que se buscará en la biblioteca el input "Boolean", al cual

se le podrá cambiar el nombre dando doble click sobre éstos para identificar la función de cada uno, que en este caso será 1) para indicar la restricción a utilizar y 2) si se desea ejecutar el análisis.

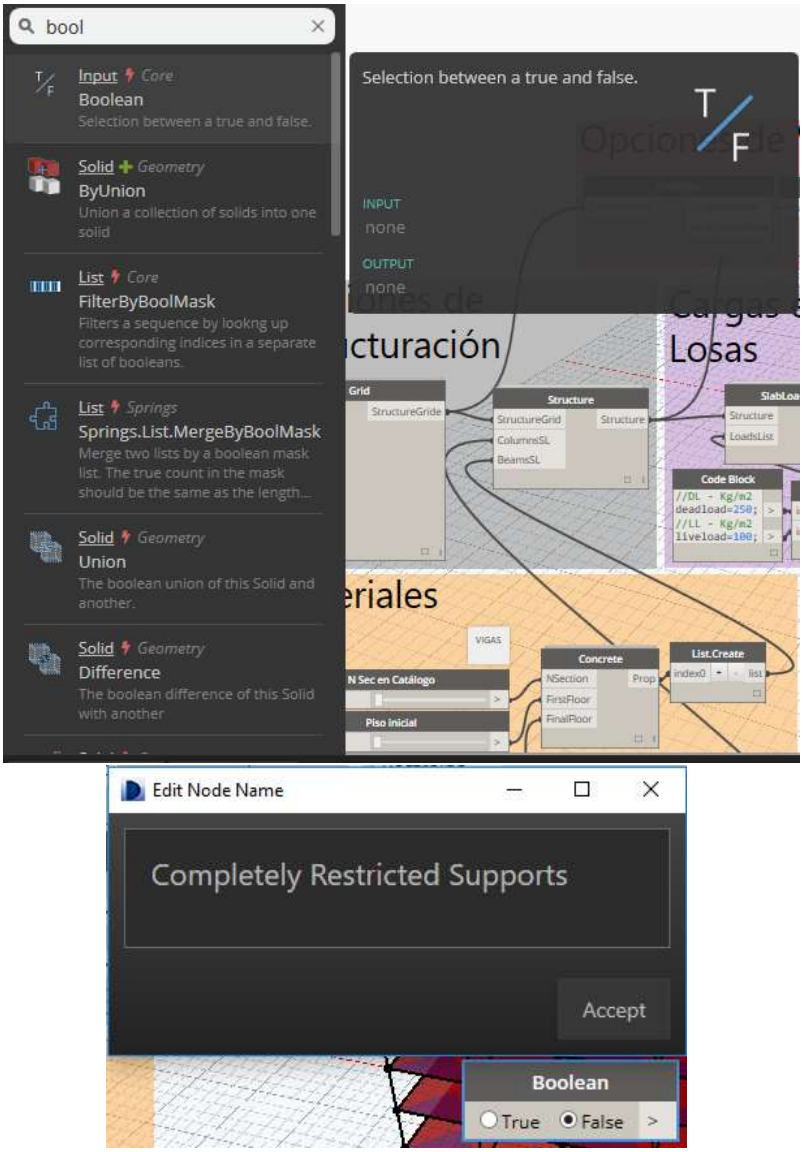


Figura B.22: Análisis Estructural. Incorporación y cambio de nombre de Boolean para ***MECA_StructuralAnalysis***.

Para que se pueda ejecutar el análisis estructural se deberá tener una conexión como la mostrada en la siguiente figura. Al realizarse, se deberá esperar un tiempo en lo que el software de análisis **MECA** se ejecuta, por lo que una ventana emergente saldrá e indicará que se realizó correctamente el análisis. Pero para poder visualizar los resultados de eficiencias mecánicas, habrá que

conectar el módulo de visualización “StrucAnalysisStrengthView”.

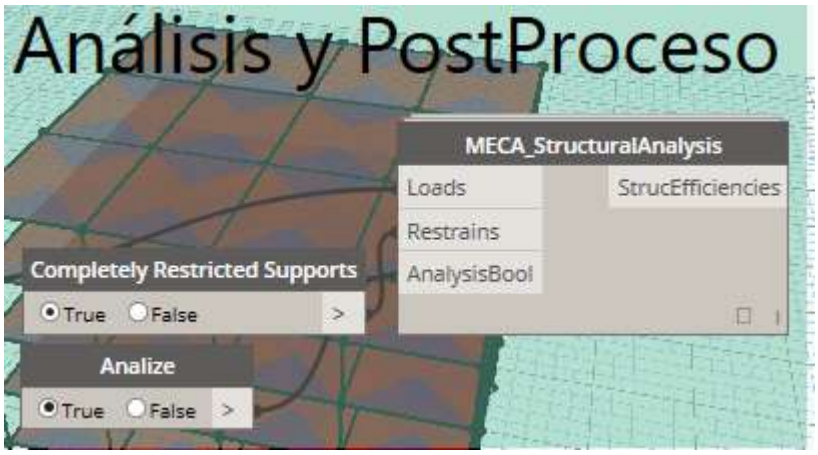


Figura B.23: Análisis Estructural. Conexión correcta para ***MECA_StructuralAnalysis***.

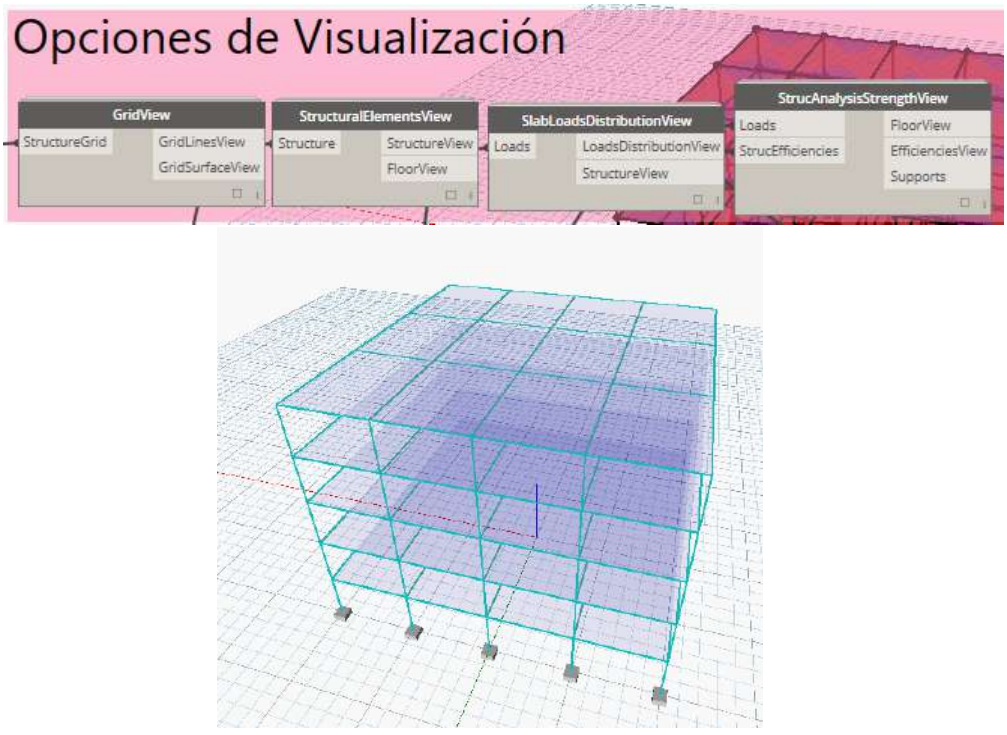


Figura B.24: Análisis Estructural. Módulo de visualización y resultado de eficiencias para ***MECA_StructuralAnalysis***.

Como podemos observar el análisis estructural de la edificación anterior da

como resultado que todos los elementos se encuentran sobre-dimensionados, por ello tienen un color azul cyan claro. Ahora, ya que se tiene la red conectada se podrá hacer cambios en cualquier punto del programa visual y actualizar de manera muy sencilla la estructuración, además de poder analizar y visualizar los resultados. La siguiente imagen muestra una nueva configuración estructural con menos separaciones entre columnas.

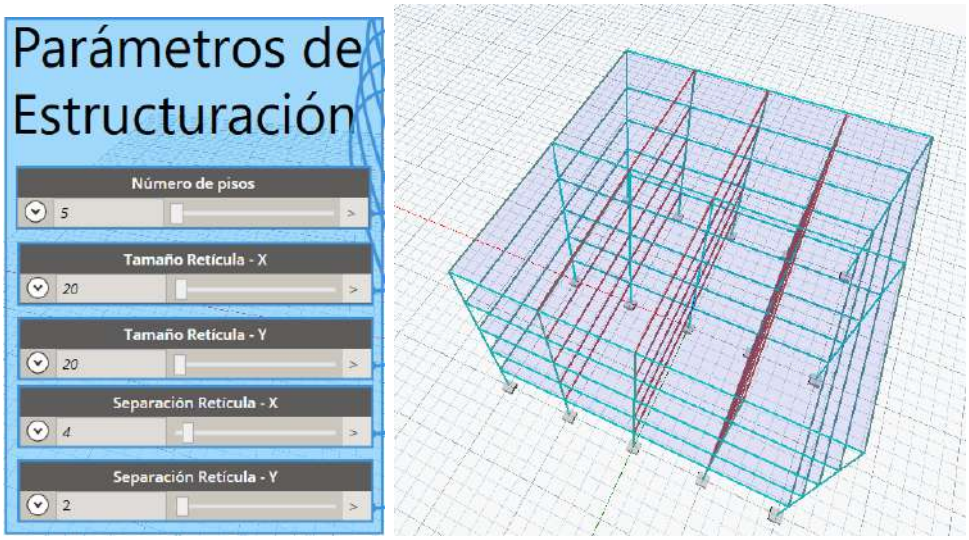


Figura B.25: Modificación de Parámetros Estructurales. Al tener conectado el programa visual, es muy sencillo modificar valores en los parámetros para obtener una nueva propuesta estructural.

La estructura mostrada antes muestra en las vigas con claros más largos que la eficiencia mecánica aumenta, es por ello que tiene un color rojo. De esta manera se pueden variar parámetros para generar múltiples propuestas estructurales. Ahora, si en lugar de modificar la estructuración, se modifican las dimensiones del sólido, la retícula sigue funcionando de la siguiente manera, por lo que la estructura se adapta al nuevo edificio modelado.

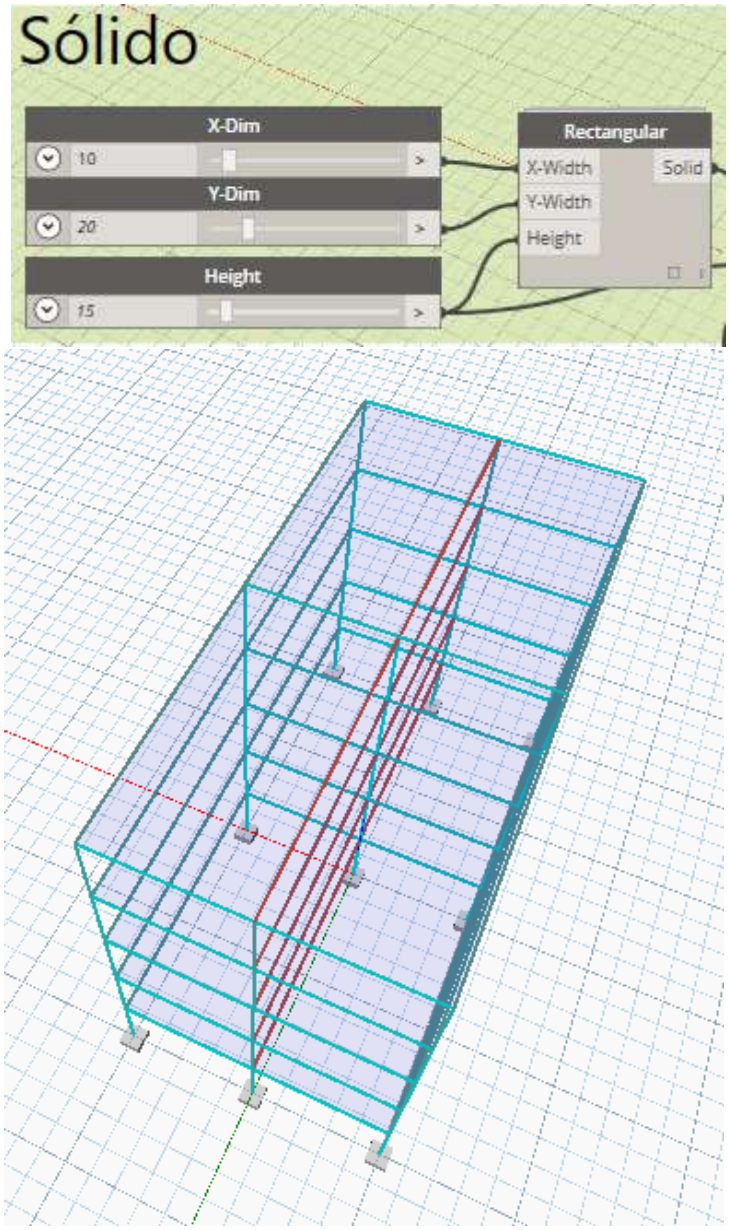


Figura B.26: Modificación de Parámetros en Sólido. Si se modifican las dimensiones del sólido tridimensional, la estructuración seguirá funcionando de la misma manera, por lo que se pueden generar más soluciones de acuerdo a los cambios que sufre un edificio..

Apéndice C

Módulos de Visualización

Además de realizar los cálculos geométricos en cada módulo de estructuración descrito anteriormente, es necesario tener un "filtro" de información que permita de manera más sencilla y amigable observar las propuestas de diseño para edificaciones. Es por ello que también se tuvo que implementar funciones en módulos que su único trabajo es tomar la información y presentarla. **Dynamo** tiene como característica que cualquier módulo cuenta con una opción de visualización o *"Preview"* que se activa al hacer click derecho sobre cada nodo y seleccionando esta opción. Para funciones como creación de rectángulos, es muy sencillo ya que al ingresar los parámetros solamente se necesita ejecutar y al activar la visualización, aparece tal cual el rectángulo. La cuestión con los módulos que se programaron es que la cantidad de información resultante es tan grande que al momento de activar la visualización normal, aparecerán múltiples geometrías y con esto es casi imposible observar la estructuración. Por ello es necesario un módulo auxiliar que al activarlo solo se presente la información deseada.

C.1. Visualizador de Estructuración

En cuanto a las funciones de estructuración se programaron dos módulos de visualización por separado, esto se debe a que al tener un sólido conceptual que se someterá al proceso de estructuración, primero se debe proponer la configuración por el usuario y esto se hace mediante el módulo ***Grid*** descrito en la sección 3.3. Debido a que se decidió optimizar el tiempo que necesita el usuario para estructurar, se utiliza un módulo para definir los parámetros y construir una retícula visible que ayuda de manera más amigable a proponer la estructura. A este módulo se le programó una función de visualización especial, donde el único propósito es generar una caja con líneas exteriores, delimitar las vigas y formar la retícula base.

La forma de programar estos módulos fue mediante redes *DAG* implemen-

tadas dentro de *"Custom Nodes"* propios de **Dynamo**, que son simplemente nodos que reducen el número de conexiones y crean un módulo más práctico para el usuario. Al seleccionar crear un nodo personalizado, se abre una ventana que contiene un espacio para crear un *DAG*, además el archivo en el cual se guarda este nodo tiene una terminación "dyf" en lugar de la extensión normal de archivo de **Dynamo**, la cual es "dyn". Lo unico que se necesita para una función de visualización es proporcionar como "input" la información generada por el módulo ***Grid***, tal como se observa en la imagen C.1. Mientras que la figura C.2 muestra este espacio de nodo personalizado, el software pinta de color amarillo el fondo del *canvas* para diferenciar del espacio normal. Para mostrar de forma gráfica los datos se utilizó la función de **Dynamo** llamada **Display.ByGeometryColor(Geometry, Color)**, donde se asignó un color verde-azul a la representación de la retícula. En lugar de solo una geometría se le proporcionó listas que contienen las líneas que forman la retícula y las cuatro superficies que forman la caja.

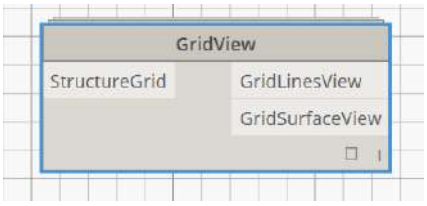


Figura C.1: Visualización Grid. Nodo personalizado ("Custom Node"), el cual se encarga de mostrar la retícula para que el usuario proponga una configuración estructural.

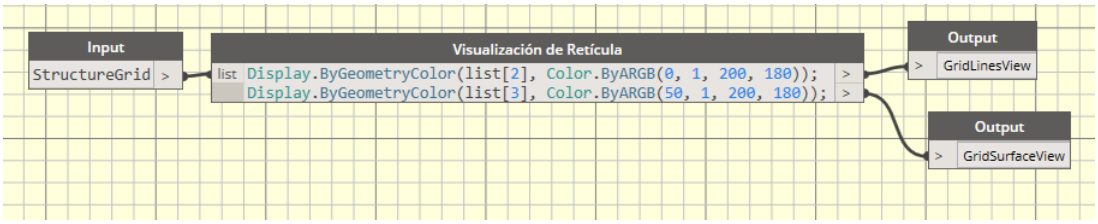


Figura C.2: Visualización Grid. Red DAG interna del nodo personalizado, este espacio sirve para reducir el número de conexiones y además para crear funciones nuevas a incorporar al software.

El segundo módulo de visualización para la sección de estructuración es requerida posteriormente a que se generan los elementos estructurales como vigas y columnas utilizando el módulo ***Structure***. Como se mencionó en dicha sección 3.5, las vigas y columnas se idealizan como líneas conectadas por nodos, ya que el modelo de análisis estructural utilizado, requiere solamente de las propiedades geométricas de la sección transversal como un dato y no es necesario modelar a detalle dichos elementos. Entonces lo único que se requiere

es tomar la información de las barras generadas por el módulo y colocarlo en un *Custom Node* para facilitar su uso.

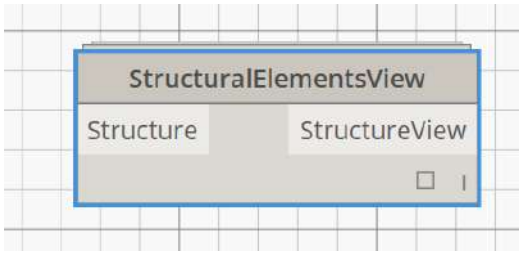


Figura C.3: Visualización Structure. Nodo personalizado ("Custom Node"), el cual se encarga de mostrar las barras y nodos que se utilizarán como elementos estructurales en la edificación.

C.2. Visualizador de Distribución de Fuerzas

Una parte indispensable en este trabajo fue la asignación de cargas a la edificación, y es por ello que se implementó el módulo ***Slab Loads*** revisado en la sección 3.6. Debido a que la distribución de cargas gravitacionales se realiza mediante un método geométrico, pareció importante crear un módulo que permitiera al usuario observar como se está asignando cada área a los elementos estructurales. Por lo tanto se creó otro *Custom Node* que recibe la información generada por éste módulo y los anteriores. La figura C.5 muestra el módulo de visualización mientras que la figura C.8, describe el procedimiento utilizado.

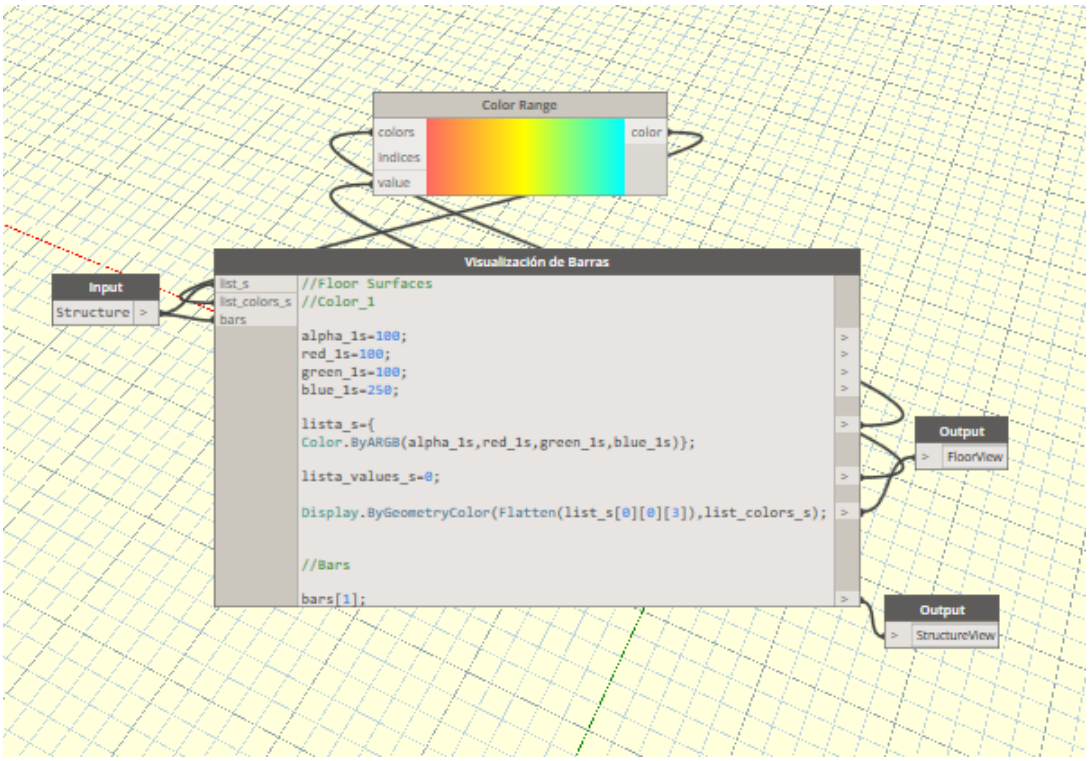


Figura C.4: Visualización Structure. Red DAG interna del nodo personalizado, en este caso se observa que se utiliza la función "Display" para visualizar tanto barras como superficies de losas.

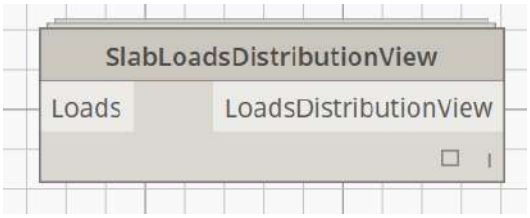


Figura C.5: Visualización Slab Loads. Nodo personalizado ("Custom Node"), el cual se encarga de mostrar las áreas tributarias generadas mediante el método geométrico de distribución.

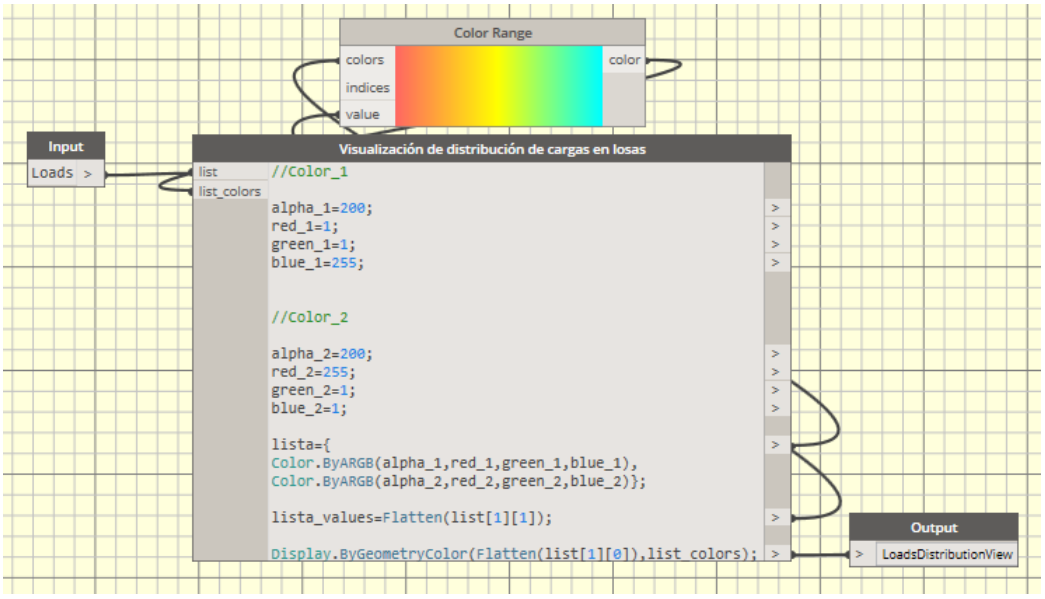


Figura C.6: Visualización Slab Loads. Red DAG interna del nodo personalizado, se utilizan dos colores para asignar las cargas. El color 1 denota al azul como mínima carga asignada mientras el rojo es la carga máxima. Utilizando la función "Color Range", se indican dichos colores y la lista con la interpolación calculada antes para asignar los colores a las superficies. Como último paso se usa "Display" para realizar la función de visualización de las superficies

C.3. Visualizador de Comportamiento Estructural

Ya que se ha realizado el análisis estructural, es necesario conocer el comportamiento de la estructura de una manera sencilla. Hasta este punto, lo único que se tiene es un archivo *.res*, con los resultados del análisis realizado por el software **MECA**. Los resultados que contiene son los desplazamientos nodales, las fuerzas en extremo de barra, las reacciones, y si se utilizó algún módulo de diseño también se presentan las resistencias de cada barra verificadas por algún código de construcción, las cuales son escritas de manera porcentual. Sin embargo abrir este archivo de texto será alarmante, ya que se presentará una cantidad de información proporcional al tamaño de la estructura. Por lo que si se analizó una edificación con varios niveles y pórticos, seguro que será prácticamente imposible evaluar el funcionamiento de la estructura. Dado esta problemática y debido a que ya no se cuenta con un *Postproceso*, como lo tenía el **MECA 1.0**, se decidió crear un módulo que recibiera esta información de resultados y la presentara de forma gráfica dentro de **Dynamo**. El módulo programado se denominó ***Post_ Pro*** y está incluido dentro del módulo de análisis.

El funcionamiento de este módulo es sencillo ya que se basa en una lectura de archivos de texto. Dado que se conoce la estructura del archivo de resultados de **MECA**, solo se implementaron ciclos que reciben la información estructural. La preguntas eran: ¿Qué resultado es el que permite al usuario observar de manera más precisa y rápida el comportamiento de la estructura? y ¿Cómo podemos saber si la estructura funciona?, hablando de términos de resistencia. Debido a que una propuesta de configuración estructural es aceptada cuando sus elementos pasan la verificación en el diseño estructural, se decidió que el resultado a presentar en **Dynamo** es el valor porcentual de resistencia obtenido de la verificación de diseño, denominado *"Eficiencia."* Entonces el objetivo de este módulo es almacenar el vector de eficiencias y asociarlo con las barras que se encuentran en el modelo. Esto es sencillo ya que de acuerdo a la sección 3.5, los elementos se organizaron de manera que primero se ingresan las vigas y posteriormente las columnas. Entonces los valores de color varían de 0 a 1, y utilizando cierta función, es posible graficar la estructura con color en las barras. El código de color empleado es el siguiente:

Color	Definición
Azul Cyan	Eficiencias bajas, de 0 % a 30 % de la resistencia disponible. Por lo que las barras están trabajando con un rango de seguridad alto y se podría decir que se encuentran *sobre-dimensionadas*.
Azul Oscuro - Morado	Eficiencias intermedias, de 30 % a 70 % de la resistencia. Los elementos se encuentran con un valor aceptable para considerar en el diseño, tienen un valor de seguridad confiable pero no están "sobradas" en cuanto al material utilizado.
Morado - Rojo	Eficiencias altas, de 70 % a 100 % de la resistencia. Los elementos en las primeras eficiencias se encuentran aún recomendables pero bajo la responsabilidad del diseñador ya que hay muchos factores que pueden resultar en una falla estructural al utilizar estas propuestas con eficiencias altas. Los valores que se acercan al 100 % aunque son más óptimos, existe mucha probabilidad de fallar al no considerar un rango de seguridad.
Blanco	Eficiencias excedidas, mayor a 100 % de la resistencia disponible. Este color indica que el elemento no cumple con la verificación de diseño, por lo que no se recomienda proponerlo en la estructura.

Tabla C.15: Tabla de relación de colores presentados por *Postproceso* en el módulo de ***Analysis*** programado en **Dynamo**.

Finalmente se exporta esa información sobre el rango de color a usar y el arreglo de barras, después una función de visualización la toma para graficar la estructura con colores. Además se añade un arreglo que manda un mensaje avisando sobre las barras que sobrepasan el porcentaje de resistencia, entonces el usuario rápidamente puede identificar cuando existen elementos estructurales que fallaran y poder proponer una nueva configuración.

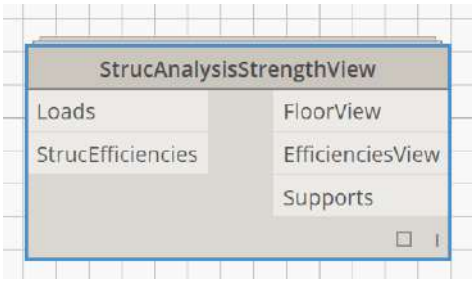


Figura C.7: Visualización Eficiencias. Nodo personalizado ("Custom Node"), su objetivo es mostrar en un código de color los resultados de resistencia de cada barra de acuerdo al análisis estructural.

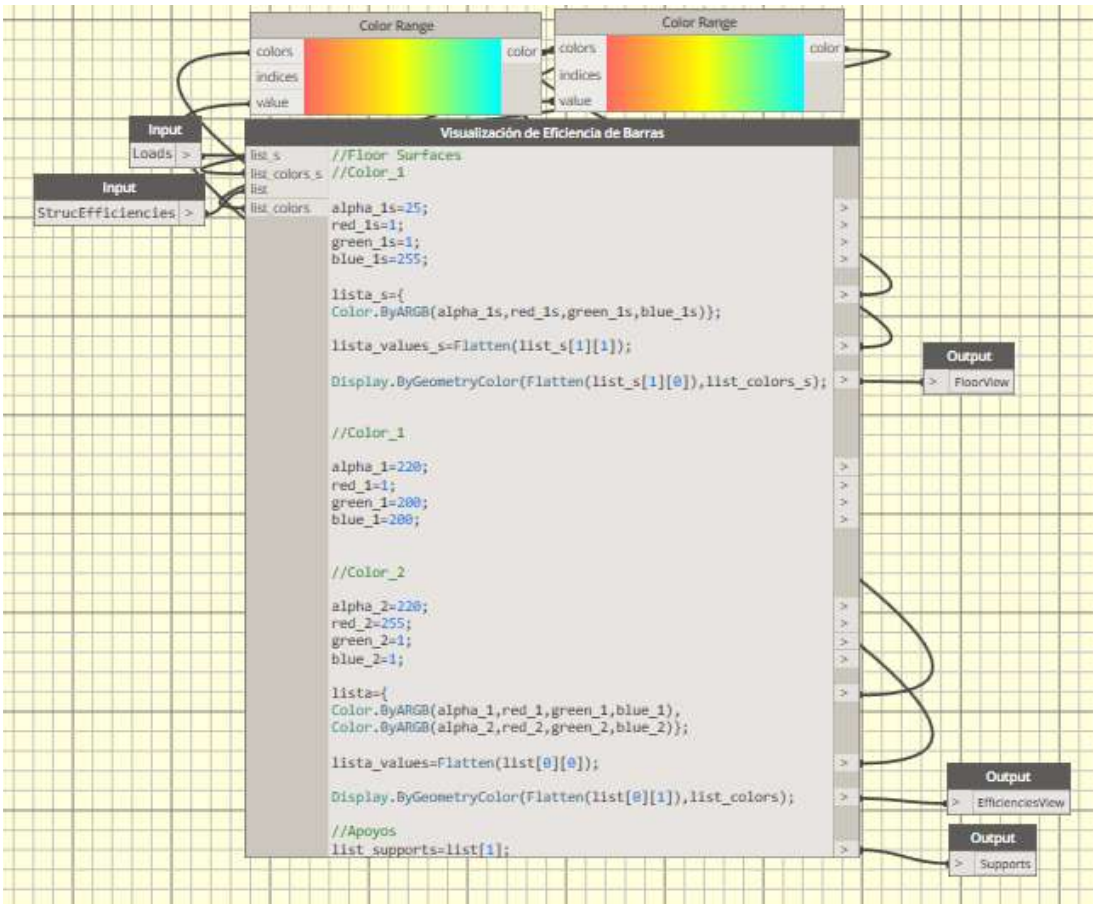


Figura C.8: Visualización Eficiencias. Red DAG interna del nodo personalizado. Se puede observar que se tienen dos secciones, la primera corresponde a la asignación de color de las superficies que se obtienen del módulo de fuerzas ***Slab Loads***. Posteriormente se observan los colores mínimo y máximo que las barras utilizaran para realizar una interpolación con cada valor de eficiencia obtenido. En ambos casos se utiliza la función "Display"

Bibliografía

[1] Aluvisa. Torre mayor. 82

[2] Architizer. Cineteca nacional siglo xxi. 6

[3] Architizer. Project liverpool department store insurgentes. 6

[4] Eastman Baer, A. y M. C., Henrion. Geometric modeling: a survey. *Computer-Aided Design*, (11):253–272, 1979. 24

[5] L. Blaise Barney. Introduction to parallel computing. 88, 89

[6] Barry. Boehm. Software engineering. *IEEE Transactions on Computers*, (12):1226–1241, 1976. 36

[7] I. Braid. *Designing with Volumes*. Cantab Press, Cambridge, UK, 1973. 24

[8] Danielmtzilluminati. torre mayor desde el castillo de chapultepec en la ciudad de mexico. 77

[9] D. Davis. *Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture*. PhD thesis, RMIT University, 2013. xviii, 26, 27, 29, 35, 36, 37, 38, 40, 41, 42, 43, 46, 47, 48, 50, 51, 167, 170, 171, 201, 204

[10] Imágenes Aéreas de México. Helipuerto torre mayor. 83

[11] Universidad Nacional del Oeste (UNO). Programación ii: Gestión automática (implícita) de memoria dinámica. 106

[12] Mexico Desconocido. Zona arqueológica el tajín. 149

[13] E. Dijkstra. Go to statement considered harmful. *Communications of the Association for Computing Machinery*, (3):147–148, 1968. 41

[14] Dynamo. Foro de dynamo. 108

[15] Nogueira F. Diseño paramétrico. 29

[16] ISO (International Organisation for Standards). Compendium of software quality standards and metrics. 48

[17] GiD. Gid: Software de pre y post - proceso. 191

[18] GitHub. Zero-touch-plugin-development. 108

[19] Vasshaug H. Håvard vasshaug blog. 31, 33, 39

[20] Vasshaug H. Dynamo for structural design. *Revit Technologic Conference*, pages 1–5, 2014. 56

[21] B. Henderson-Sellers y Tegarden D. The theoretical extension of two versions of cyclomatic complexity to multiple entry/exit modules. *Software Quality Journal*, (4):253–269, 1994. 50

[22] Shah J.J. y Mäntilä M. *Parametric and feature based CAD/CAM: concepts, techniques, and applications*. IWiley-Interscience publication, Canada, 1995. 29

[23] Y. Kalay. *Modeling Objects and Environments*. Wiley, New York, NY, 1989. 30

[24] Luke Kennedy. A brief history of autocad. 23

[25] Apostolos Konstantinidis. Load path from the structure's slab to the ground. 134

[26] A. Leitão, Santos L., y Lopes J. Programming languages for generative design: A comparative study. *International Journal of Architectural Computing*, (1):139–162, 2012.

[27] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, College Park, MD, 1988. 24

[28] Torre Mayor. Características de la torre mayor. 77

[29] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (2):308–320, 1976. 41, 50

[30] Culture Trip Mexico. Why soumaya is a mexico city must see. 6

[31] B. Meyer. *Object-Oriented Software Construction. Second edition*. Prentice Hall, Upper Saddle River, NJ, 1997. 40, 41, 51

[32] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, (12):1053–1058, 1972. 41

[33] Dynamo Primer. Manual de usuario sobre el software **Dynamo**. 43, 44, 45, 46, 52, 53, 54, 55, 57, 58, 60, 61, 63

[34] A. Requicha. Representations of rigid solids: theory, methods and systems. *ACM Computer Survey*, (12):437–466, 1980. 30

[35] A. Requicha y H. Voelcker. Solid modelling: current status and research directions. *IEEE Computer Graphics and Applications*, (3):25–37, 1983. 24

[36] Mimoa Images Resources. Centre pompidou-metz.

[37] D. Rutten. Programming, conflicting perspectives. i eat bugs for breakfast. 37

[38] Botello S., Muttio E., y J. Salazar. *Análisis Estructural Estático y Dinámico. Armaduras, Vigas y Marcos.* Universidad de Guanajuato, Guanajuato, México, 2017. 152

[39] Botello S., Marroquin J. L., Rionda A. B., y R. Ducoing. *MECA Programa para el Análisis Matricial de Estructuras.* Universidad de Guanajuato, Guanajuato, México, 1997.

[40] R. Sacks y R. Barak. Impact of three-dimensional parametric modeling of buildings on productivity in structural engineering practice. *Automation in Construction*, (17):439–442, 2007. 10, 12, 14

[41] R. Sacks, C. Eastman, y G. Lee. Parametric 3d modeling in building construction with examples from precast concrete. *Automation in Construction*, (13):291–312, 2003. 14, 24, 25, 30, 32, 203

[42] J. Salazar. *Metodología Educativa de la Materia Análisis Estructural 2.* Master's thesis, Escuela de Ingeniería Civil, Universidad de Guanajuato, 2013. 150, 151, 153, 154, 156

[43] F. Scheurer y Hanno S. Lost in parameter space ? *Architectural Design*, pages 70–79, 2011. 28

[44] J. Sharp. *A Brief Introduction to Data Flow. In Data Flow Computing: Theory and Practice.* John Sharp, Norwood, NJ, 1992. 40

[45] Assistant Engineer on 30 Jan 1896 Signed: E A Cullen. Lighthouse for caloundra head. 21

[46] Rick Smith. Technical notes from experiences and studies in using parametric and bim architectural software. 28

[47] A. Sorensen. Impromptu : An interactive programming environment for composition and performance. *In Generate and Test: Proceedings of the Australasian Computer Music Conference 2005*, pages 149–154, 2005. 43

[48] I. Sutherland. *Sketchpad: a man-machine graphical communication system.* PhD thesis, Massachusetts Institute of Techonology, 1963. 23, 30

[49] Dassault Systemes. Lanzamiento de catia versión 3. 25

[50] Tombs y Memorials Conservation Project. St james west littleton. 21

[51] B. Victor. Inventing on principle. presentation at turing complete: Canadian university software engineering conference, montreal, 20 january. 41

[52] A. Watson y T. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* National Institute of Standards and Technology, Gaithersburg, MD, 1996. 51

[53] Arquitectura Obras Web. Supermercado chedraui. 6

[54] E. Weisstein. *CRC Concise Encyclopedia of Mathematics.* Chapman & amp; Hall/CRC, Boca Raton, Florida, 2003. 29

[55] E. Weisstein. *Information Technology: Software product quality – Part 1: Quality model. ISO/IEC 9126.* American National Standards Institute, New York, NY, 2003. 48

[56] Arie Willem de Jongh. Create an organic roof structure using parametric design, centre pompidou-metz [designplaygrounds]. 29, 34

[57] G. Wyatt. Maintaining bim integrity in the structural engineering office. *Autodesk*, pages 1–8, 2007. 7